금융 문제에 대한
병렬 몬테카를로 시뮬레이션

유 현곤
연세대학교 수학과 박사과정

Email : yhgon@yonsei.ac.kr

2008년 7월 23일 수요일 10시
고려대학교 금융 세미나

연세대학교
YONSEI UNIVERSITY
금융퀀트연구센터

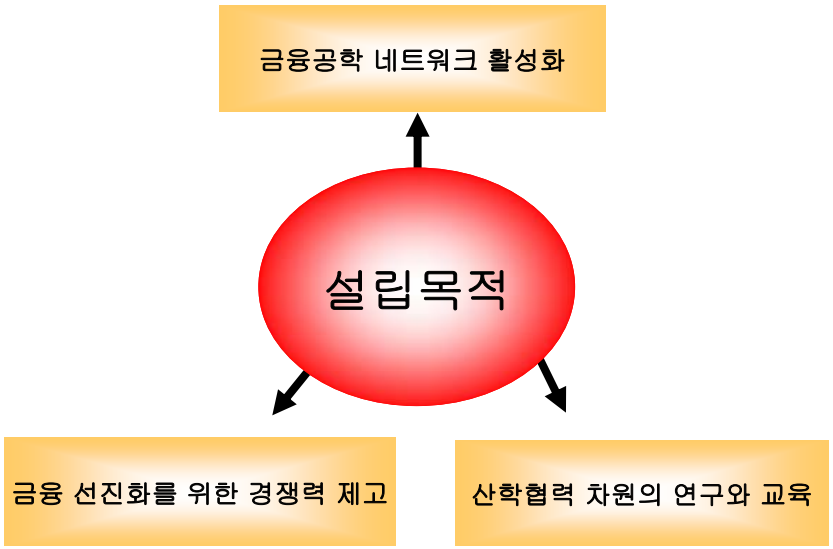
Agenda

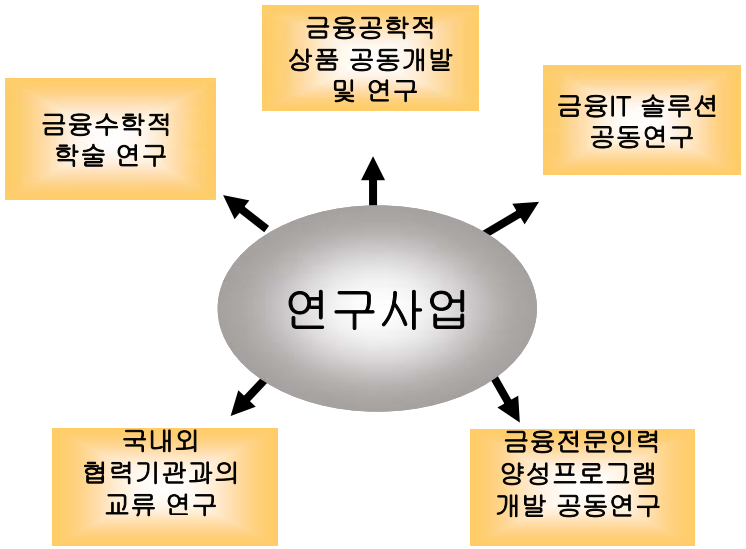
1. 연세 금융퀀트연구센터 소개
2. 기초 금융과 Monte Carlo Simulation
3. 병렬처리(pMC)에 대한 이해
4. 수퍼컴퓨터를 이용한 pMC I (openMP)
5. 수퍼컴퓨터를 이용한 pMC II (MPI)
6. 대안적 방법론 1 Clearspeed 가속보드를 이용한 pMC
7. 대안적 방법론2 CUDA를 이용한 pMC
9. Q & A

연세대학교 금융퀀트연구센터

http://quant.yonsei.ac.kr/

교수님 3분, 박사과정 : 4명, 석사과정 : 12명, 학부생 : 15명

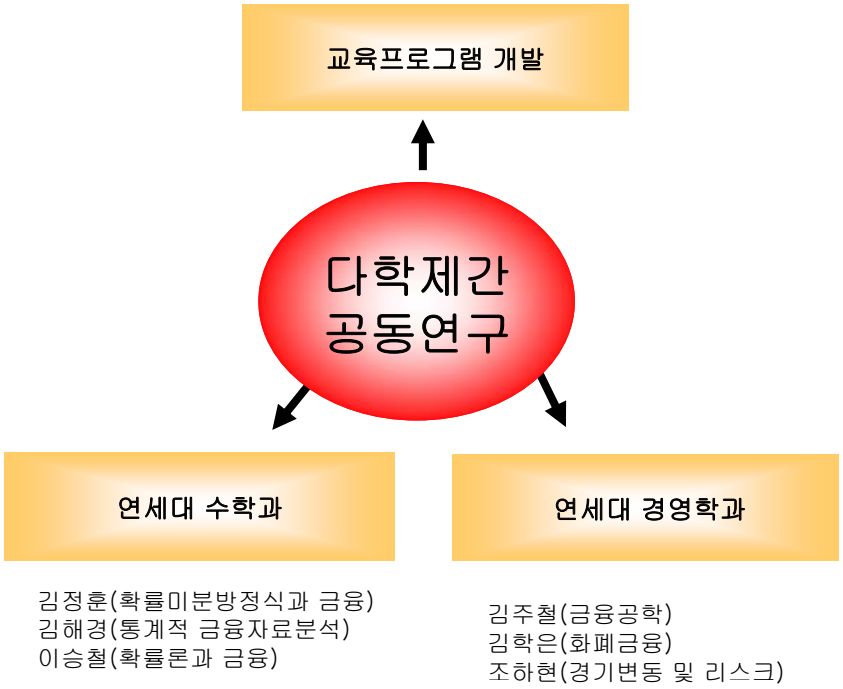


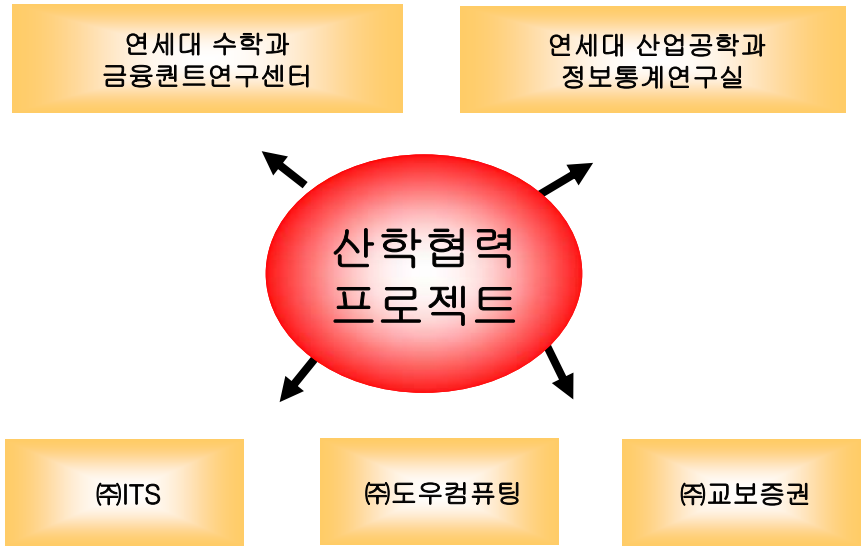


국내외 학술교류
해외 석학 초청 세미나 – 2008년 5회 개최
산학협력 세미나 – 2008년 7회 개최
병렬 컴퓨팅 자문 – LG전자, KISTI,
산학협력 공동 연구 프로젝트 등 – ㈜도우컴퓨팅, ㈜ITS, ㈜교보증권

연세대학교 금융퀀트연구센터

공동연구





연세 금융퀀트 연구센터 연구 현황

연구센터 정식 설립 인가는 2008년 3월

하지만 1997년 초부터 3분의 교수님들을 중심으로 금융수학 연구 시작

금융수학 교재 출판
이승철, 수학과 현대금융사회, 2002, 교우사
김정훈, 금융수학, 2005, 교우사
김정훈, 금융과 수학의 만남, 2006, 교우사

1998년부터 SDE 강의 (대학원, Oksendal 교재)
2000년부터 수학과 현대사회 강의 (학부 교양과목)
2004년부터 응용수학(학부 금융수학 강의)
2006년부터 Levy process 강의 (대학원, Applebaum 교재, Cont 교재)

연세 금융퀀트 연구센터 연구 현황

금융관련 해외 학술 교류

2004년 호주

2005년 교토

키지마 교수 초청

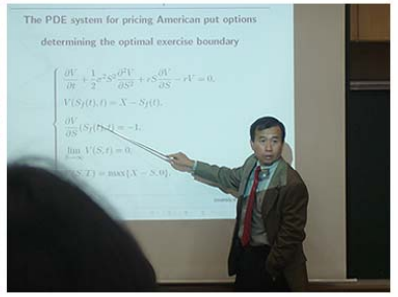
2005년 호주

연세 금융퀀트 연구센터 연구 현황

해외 학술 교류

2006년 Bacheiler Conference 참석

2008년 American Option의 대가
호주 Zhu 교수님의 세미나

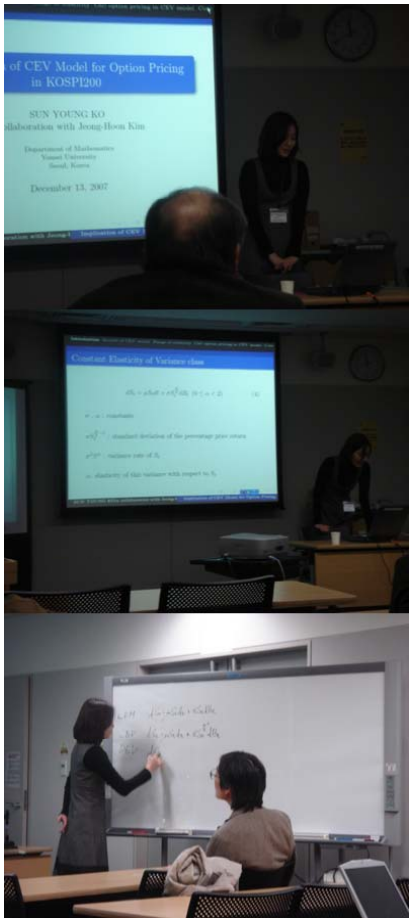


연세 금융퀀트 연구센터 연구 현황

해외 학회 발표

2007년 게이오 대학

SV bond option



CEV 모델

SVRS 모델

연세 금융퀀트 연구센터 연구 현황

국내 학회 참석

2008년 금융공동학회 참석

2008 kms 미팅 (계명대학교)
2008 Kms 확률론워크숍 (고려대학교)
2008 16차 ICFIDCAA (동국대학교)

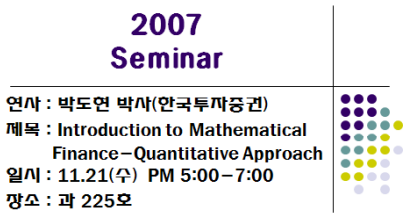
연세 금융퀀트 연구센터 연구 현황

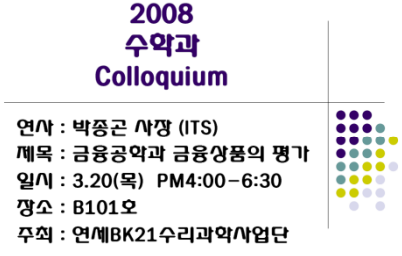
산학협력세미나 개최

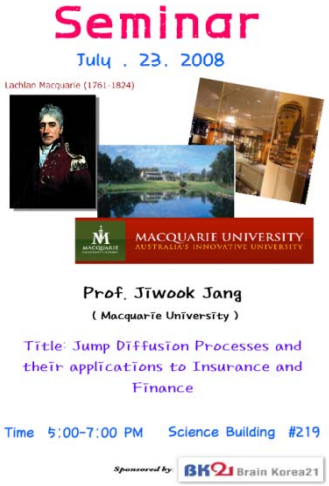
이하 생략
2007년 6월 10일 장원재 박사 – 삼성증권
2007년 10월 11일 정대용 박사 – 한국금융연수원
2007년 11월 14일 서승석 박사 – 한국투자증권
2007년 11월 21일 박도현 박사 – 한국투자증권
2008년 3월 20일 박종곤 박사 – ITS
2008년 4월 16일 배원성 박사 – 도우컴퓨팅
2008년 5월 17일 김영성 과장 – 신한은행
2008년 5월 30일 김종훈 과장 – 한화증권 금융공학팀
2008년 7월 4일 Eric Young – 미국Nvida 본사
2008년 7월 11일 기호삼 박사 – KIS채권평가

연세 금융퀀트 연구센터 연구 현황

세미나 관련 포스터들



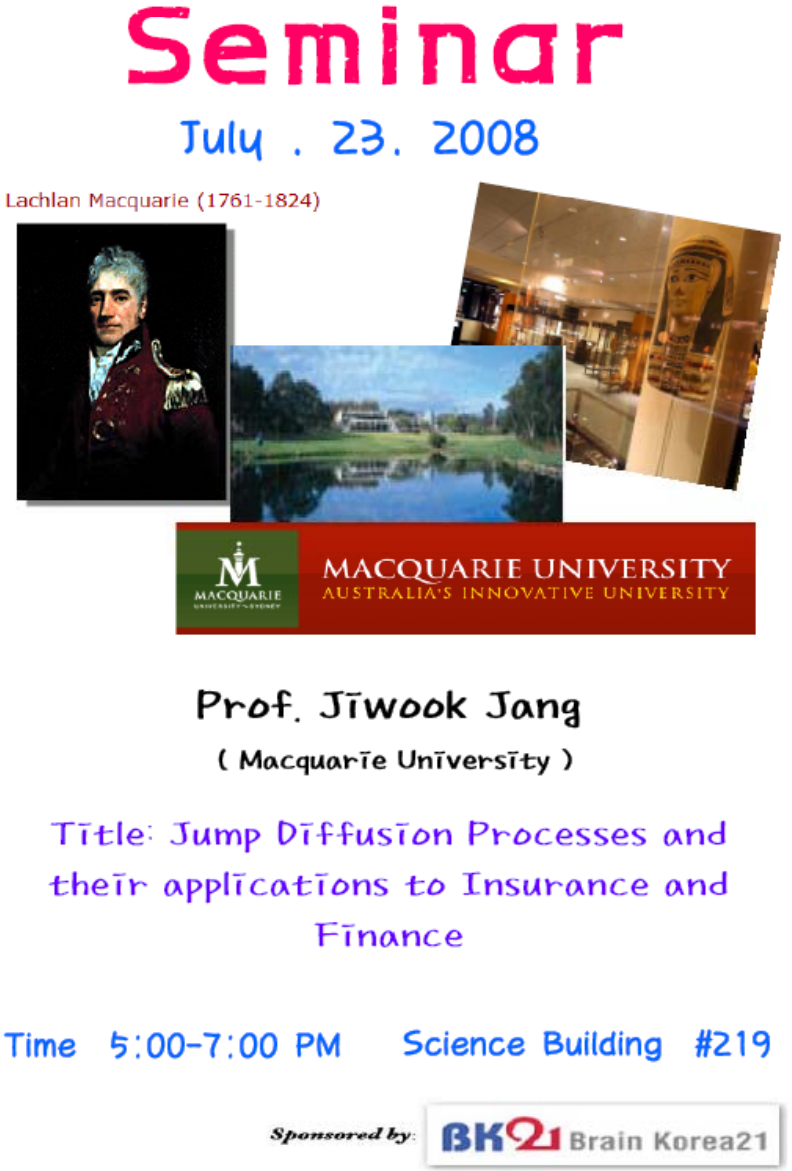




연세 금융퀀트 연구센터 연구 현황

연세대 세미나

7월 23일 저녁 5시
호주 머큐리대학
장지욱 박사님



연세 금융퀀트 연구센터 연구 현황

병렬 컴퓨팅 연구 현황

수퍼컴퓨터를 이용한 병렬처리 (Monte Carlo Simulation) - 2006년 3월부터

Clearspeed를 이용한 병렬처리 (Monte Carlo Simulation) - 2007년 6월부터
도우컴퓨팅과 공동연구 진행중

CUDA를 이용한 병렬처리 (Monte Carlo Simulation) - 2008년 1월부터
세미나 개최 - 2008년 6월 12일, 7월 4일, 7월 11일
연구 결과 발표 - KMS 확률론 워크숍 6월 13일(서울), 16th ICFIDCAA 7월 30일(경주, 예정)
ieee submit 예정

CUDA 등 병렬처리 관련 대학 연구팀 , 연구기관 및 업체들과
산학연 협력체제 및 네트워크 구축 중

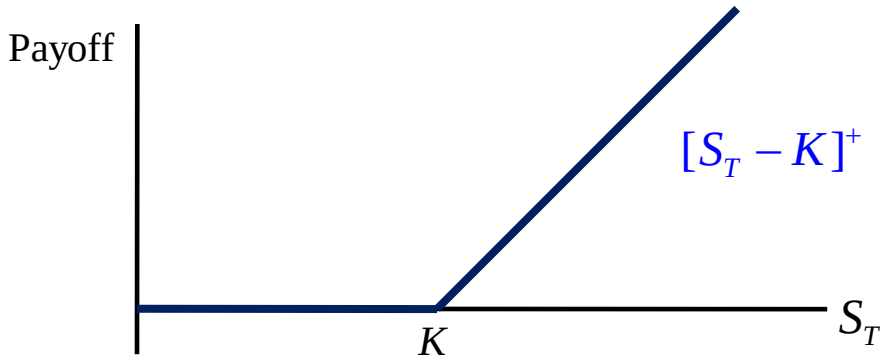
2. 기초 계산 재무 및 MC

Introduction for Option Pricing

European Call Option

1. asset dyanmics GBM : $dS_t = \mu S_t dt + \sigma S_t dW_t$

2. payoff $[S_T - K]^+$



3. option price is $C_t = E^Q[e^{-r\tau}[S_T - K]^+]$

옵션의 현재가치 $C_t = E[e^{-r\tau}[S_T - K]^+]$

직접 풀이 Closed form solution

$$C_t = S_0\Phi(d_1) - Ke^{-r\tau}\Phi(d_2)$$

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-x^2/2}dz \quad d_1 = \frac{\ln(S_0/K) + (r - \sigma^2/2)\tau}{\sigma\sqrt{\tau}} \qquad d_2 = d_1 - \sigma\sqrt{\tau}$$

PDE방법 in FDM, FEM, Meshless

$$\frac{\partial f}{\partial t} + rx\frac{\partial f}{\partial x} + ry\frac{\partial f}{\partial y} + \frac{1}{2}\sigma_x^2\frac{\partial^2 f}{\partial x^2} + \rho\sigma_x\sigma_y\frac{\partial^2 f}{\partial x\partial y} + \frac{1}{2}\sigma_y^2\frac{\partial^2 f}{\partial y^2} = rf$$

시뮬레이션 방법 Monte Carlo

$$E[e^{-r\tau}[S_T - K]^+] = \lim_{N\to\infty}\frac{1}{N}\sum_{i}^{N} e^{-r\tau}[S_{T_i} - K]^+$$

Closed Form solution

European Call Option

$$C_t = E^Q[e^{-r\tau}[S_T - K]^+]$$

4. By Ito's lemma

$$S_T^{\ Q} = S_0 e^{(r-\frac{1}{2}\sigma^2)\tau+\sigma W_\tau}$$

$$S_T = S_0 e^{(r-\frac{1}{2}\sigma^2)\tau+\sigma\sqrt{\tau}N(0,1)}$$

5. By Feynmann-Kac Theorem

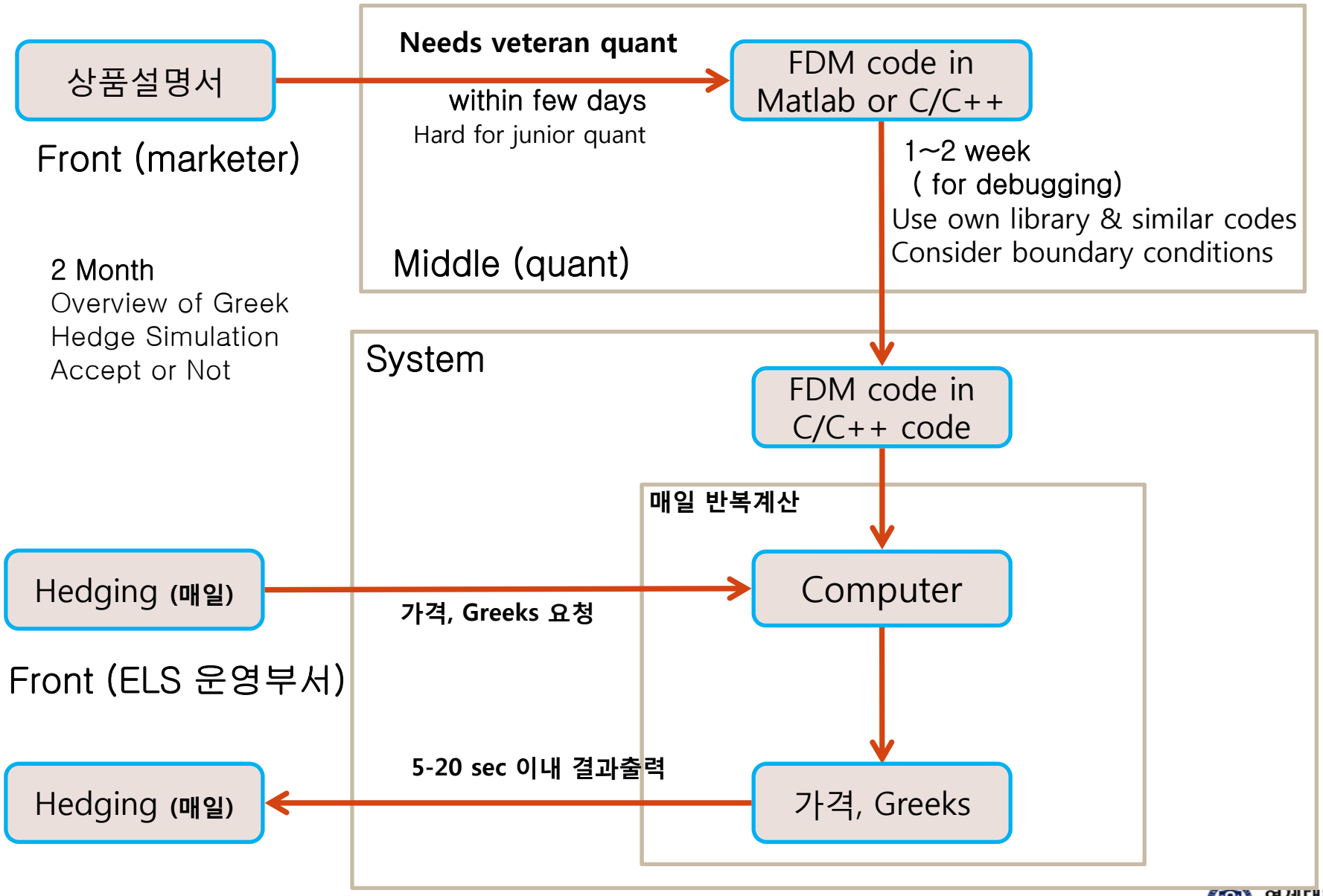
BS-PDE : $f_t + rxf_x + \frac{1}{2}\sigma^2 x^2 f_{xx} = rf$

6. Closed Form Solution

$$C_t = S_0\Phi(d_1) - Ke^{-r\tau}\Phi(d_2)$$

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-x^2/2}dz \qquad d_1 = \frac{\ln(S_0/K) + (r-\sigma^2/2)\tau}{\sigma\sqrt{\tau}} \qquad d_2 = d_1 - \sigma\sqrt{\tau}$$

새로운 구조의 상품에 대한 분석작업 (FDM)

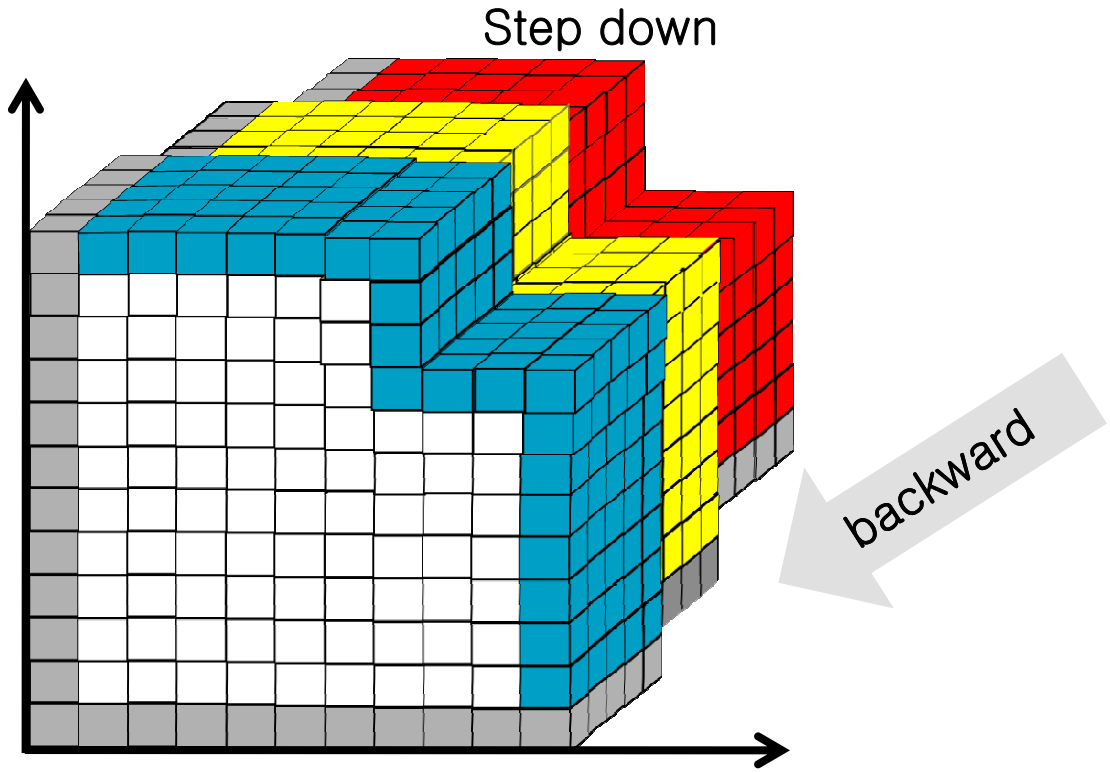


PDE method for ELS

$$\frac{\partial f}{\partial t} + rx\frac{\partial f}{\partial x} + ry\frac{\partial f}{\partial y} + \frac{1}{2}\sigma_x^2\frac{\partial^2 f}{\partial x^2} + \rho\sigma_x\sigma_y\frac{\partial^2 f}{\partial x \partial y} + \frac{1}{2}\sigma_y^2\frac{\partial^2 f}{\partial y^2} = rf$$

With Boundary Value& Terminal Payoff

Domain for 2-Star n-chance Stepdown ELS



Monte Carlo Simulation

Law of Large Number : we can compute expectation

$$\mathrm{E}[e^{-r\tau}[S_T - K]^+] = \lim_{N\to\infty} \frac{1}{N} \sum_{i}^{N} e^{-r\tau}[S_{T_i} - K]^+$$

We can generate any process S_T from given dynamics

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$
$$dS_t = \mu S_t dt + f(Y_t) S_t dW_t$$

Monte Carlo Simulation

Why MC is need ?

- Easy to imply 적용이 쉽다.

- Complex structure : path-dependence, prepayment, complex payoff, etc.
  ELS 등 복잡한 상품의 적용에 유리

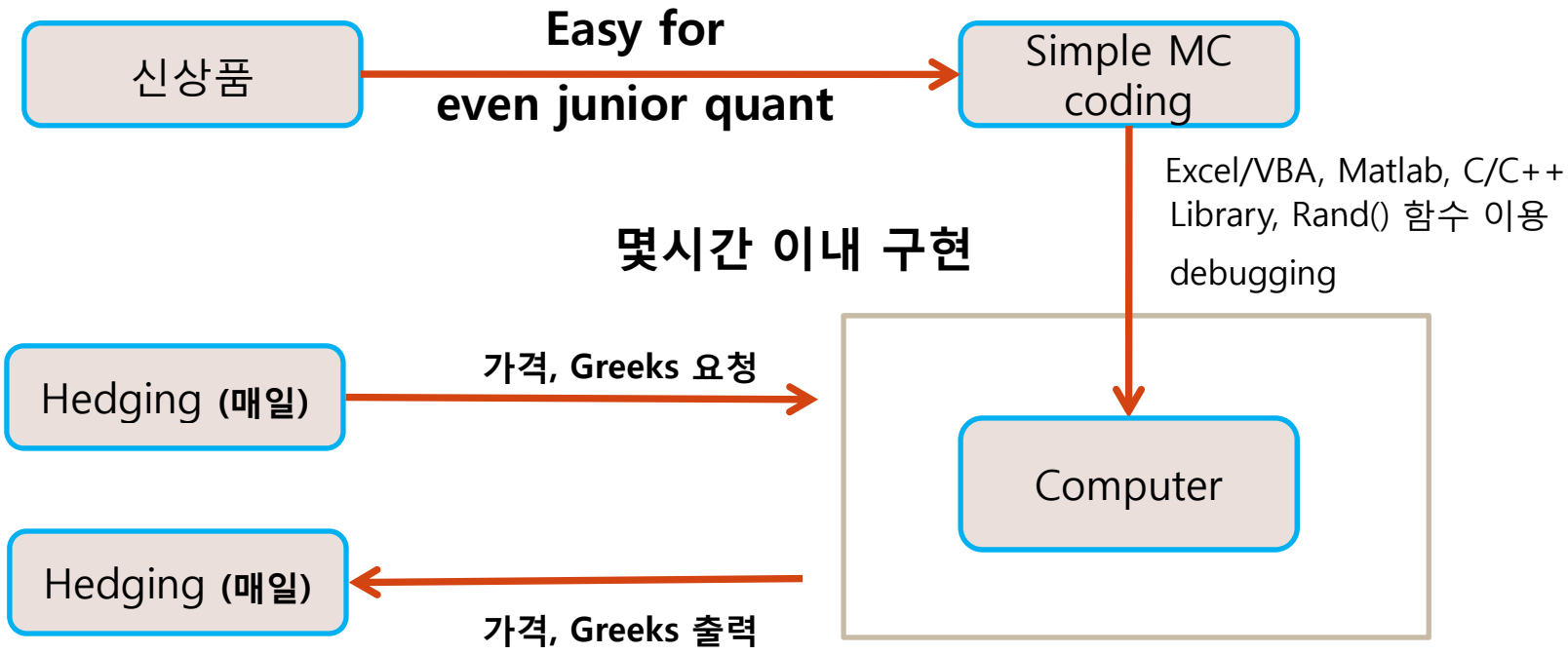
- High demension : multi-asset problems (n>4)

  ➔ MC is the only solution (or sparse grid method)
  다자산 상품에 대한 유일한 방법임

Monte Carlo Simulation

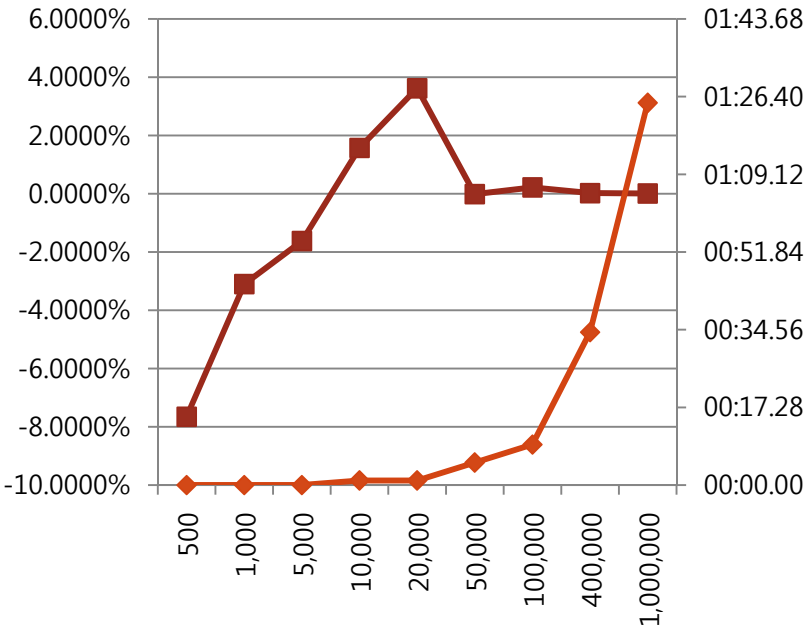
상품 발행시
Pricing
Overview of Greek
Hedge Simulation
조기상환 확률계산

상품거래시
Pricing
Greeks Search
VaR계산, 위험관리



Monte Carlo Simulation

$$E[e^{-r\tau}[S_T - K]^+] = \lim_{N\to\infty} \frac{1}{N} \sum_{i}^{N} e^{-r\tau}[S_{T_i} - K]^+$$



N : 10만번 이상 시행해 줘야함

실제 거래되는 ELS의 계산시
10만 번 실행에 1분 정도 계산시간 소요

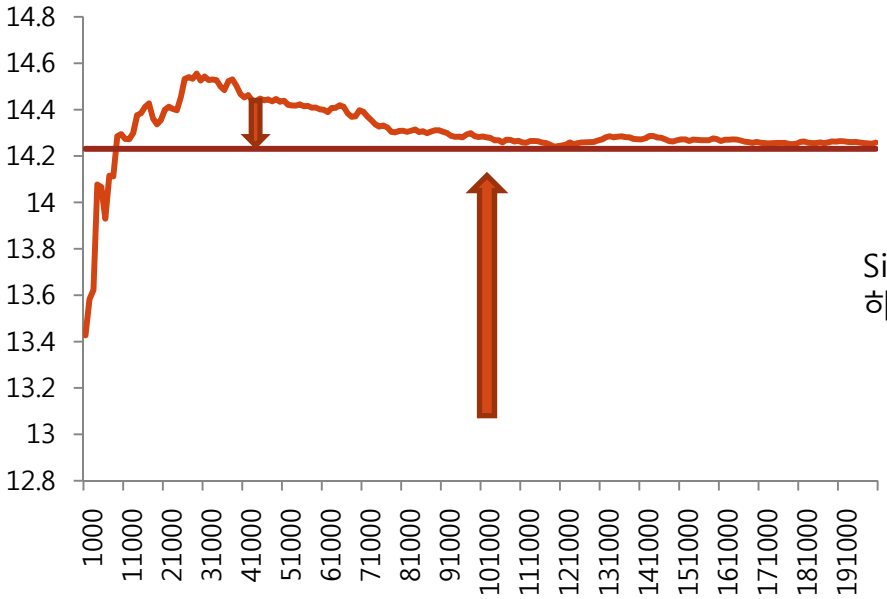
100개의 상품이라면 2 시간 이상 소요

현재 국내업계의 구현방법
1. 5만번 정도만 돌림
2. FDM 등의 방법 이용

250 time step Path simulation

환경 : Intel CPU 3Ghz

Black-Scholes Clsoed : 14.23124493415722500
Monte Carlo Simulation : 14.26851272608143400
Error : 0.03726779192420970



Simple MC의 경우10만 이상이 Simulation 하는 것이 적당 7.65sec
하지만 계산시간의 제약으로 약 5만번 정도만 돌리면 . . .

Why MC is not used ?

- Most contracts(95%) can be solved without MC. (closed, FDM)

- **Computation Time** : Too slow to get accurate results
  -1 minutes for each pricing. 4 minutes for greeks
  100 contracts : 7 hours to comutes (1 day : 6 hours)
  시나리오분석용 Greek plot 그리려면 하루종일걸림

병렬화

- **Unstable sensitivity : non-smooth greeks plot**

Reasonable Solutions to speed up MC

1. New theory in convergence
   Malliavin Calculus, Operator Technique, Asymptotics

2. Fast pseudo & **quasi-RNGs**, Transformation

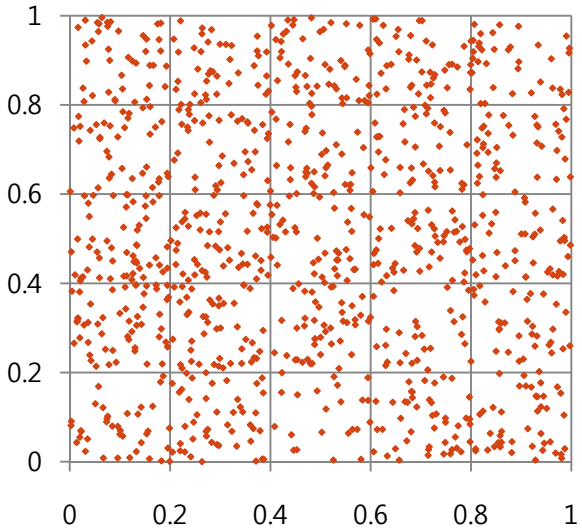
3. Control Variate, **Variance Reduction**

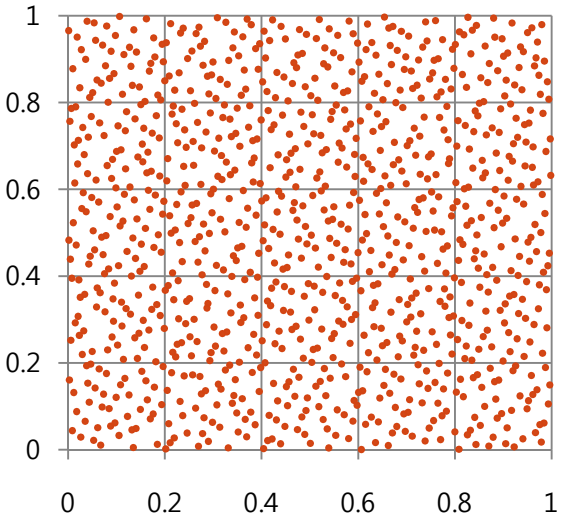
4. Using Powerful Computer

5. **Parallel Computing** 1주제
   HPC(use many CPUs) - **OpenMP(SMP), MPI (Cluster)**
   Alternative Method - IBM Cell BE, ClearSpeed, **GPGPU(CUDA)**
   2주제 3주제

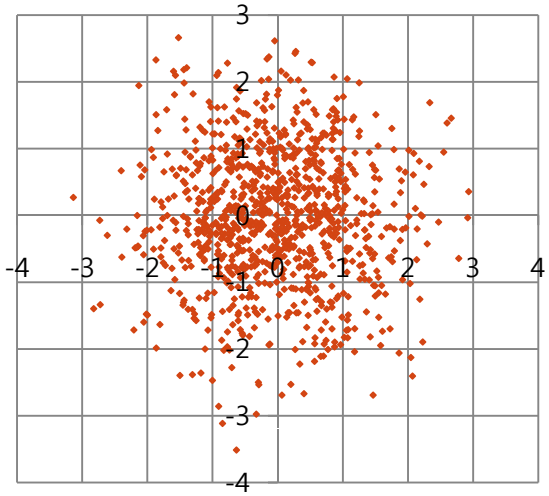
이상적 모델 : 1,2,3.4를 먼저 실시하고 5에 관심을 갖는다.

Pseudo vs. Quasi

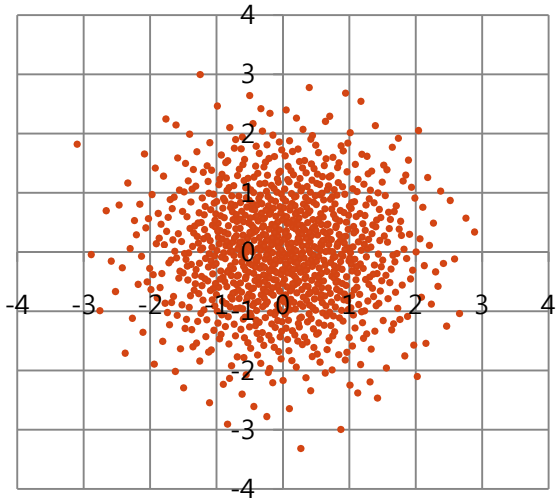




본 발표에서는 생략



LCG32



Halton Sequence

연세대학교 YONSEI UNIVERSITY
금융퀀트연구센터

3. 병렬처리(pMC)에 대한 이해

HPC와 분산처리의 차이

병렬처리는 크게 3가지로 나뉘는데

하나의 방법은 HPC로, 하나의 작업을 여러 개로 쪼개어 계산하도록 함으로써 계산시간을 단축시킴

다른 하나의 방법은 여러 개의 작업을 여러 개의 서버에 분산처리시켜 전체작업시간을 단축시키는 기법

웹서버, 게임서버에 사용되는 로드벨런싱도 분산처리기법의 하나임

계산의 관점에서 병렬처리는 HPC를 의마함

분산처리의 예
하나의 자산당 200초 걸리는 자산 200개가 있다. 총 40000초 걸림
200개의 서버에서 각 자산 정보를 보내서 돌린 결과를 받음
→병렬처리 할 필요없이 200초만에 모든 VaR 계산
Quant에게 유리(다시 코딩할 필요 없음), 회사입장 비용문제(수십억원)

금융권에서의 병렬화 진행상황

일반 프로그래밍

Excel/VBA 혹은 C/C++ 금융 프로그래밍
Excel2007에서 excel 내장 함수가 아닌 Excel/VBA, C/C++을 사용하면 오히려 속도가 느려지는 현상이 발생하는 경우가 있음 (excel에서 cell단위 자동분산처리함)

분산처리

여러명이 동시에 작업을 수행함, 혹은 여러대의 컴퓨터가 동시에 여러작업을 수행함
서버가 알아서 처리하므로 Quant는 신경쓸 필요 없음
현재 많은 중소형 증권사 (4년 전 대부분 금융권 – HPC컨퍼런스 Intel 한국지사 부사장曰)

HPC

Cluster기반 병렬시스템을 구축하였거나 구축하려는 추세 (ELS 라이센싱 대형 증권사)
외산 병렬 시스템 도입을 고려 (대형 은행)
병렬 프로그래밍이 가능한 퀀트 필요 – 인력이 절대적으로 부족함
금융관련 대학 : -- 연세대 수학과(cluster구축) 유일
비금융 대학 : 타 전공은 많은 편임 (CS, 기계 등)

대안적 가속처리 (IBM Cell, CS, CUDA 등)

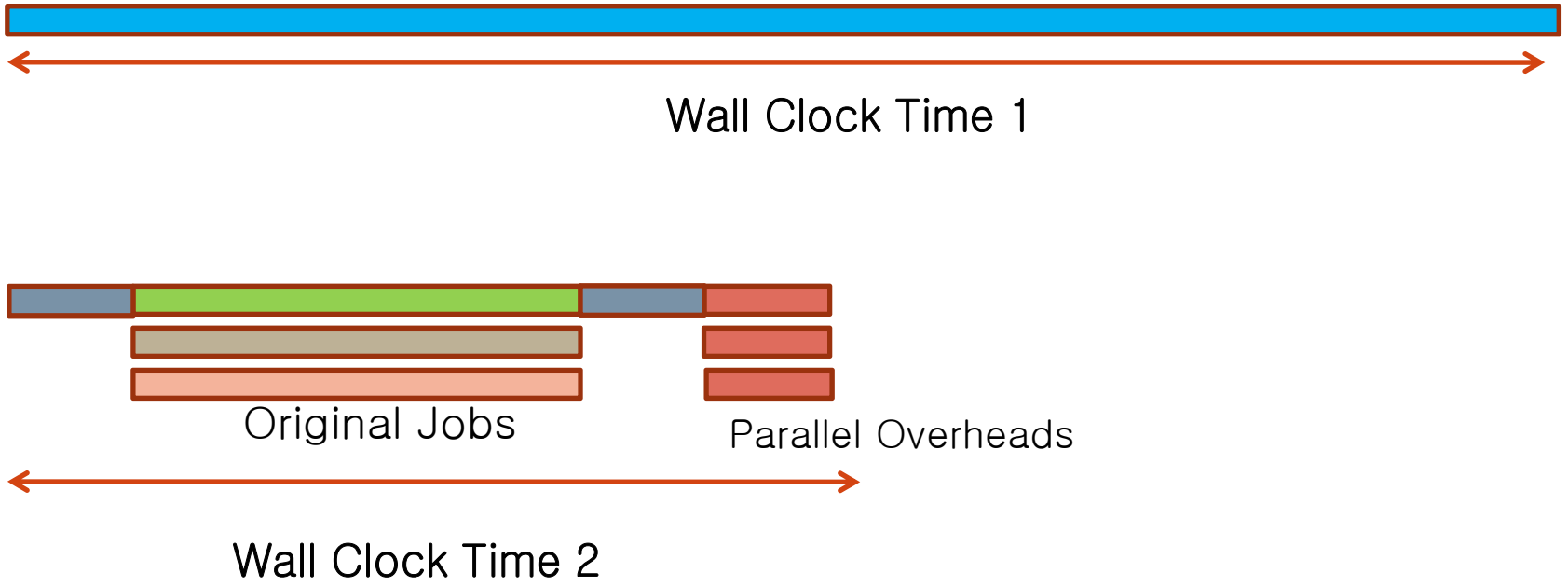
최근 관심이 높아짐, 하지만, 역시 인력부족
금융관련 대학 : 연세대 수학과 유일 (CS, CUDA), 해외(Oxford Univ.)
비금융 대학 : 이화여대 그래픽연구실(CUDA) , 각대학 컴공, 물리, 천문대기, 기계 등

병렬처리(HPC – High performance Computing)

하나의 작업을 여러 개의 CPU(Cores)로
작업을 실행시켜 계산시간을 줄이는 방법

Wall Clock Time

Task를 병렬처리 에 의해 처리하는데 걸린 전체 시간



몇 년 기다리면 빠른 컴퓨터가 나오지 않을까?

그냥 엔터치면 결과값이 나올텐데..
지금 병렬화를 배워서 금융에 적용해야하나?

현재 출시되는 모든 CPU는 Dual Core, Quad코어 기반임

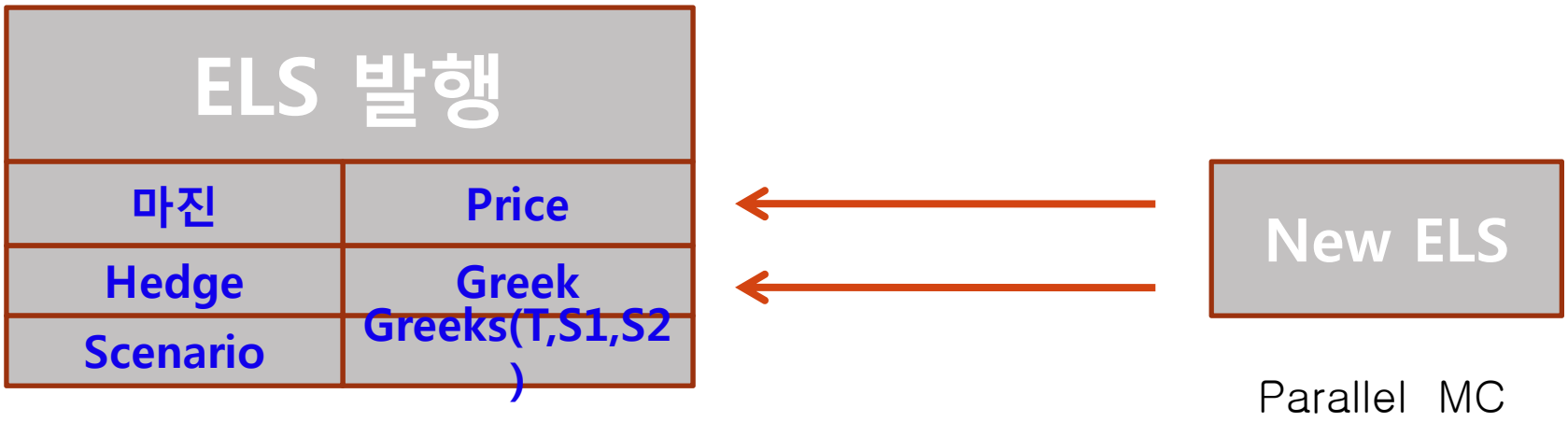
추후 출시되는 CPU는 8 core, 16 core, 32 core로
연산속도 증가보다는 하나의 칩에 코어의 개수를
늘리는 방식으로 개발되고 있음

이러한 멀티코어 시스템의 성능을 100% 발휘하도록 하려면

모두 병렬 프로그래밍을 해주어야 함.

Application for Massively Parallel MC

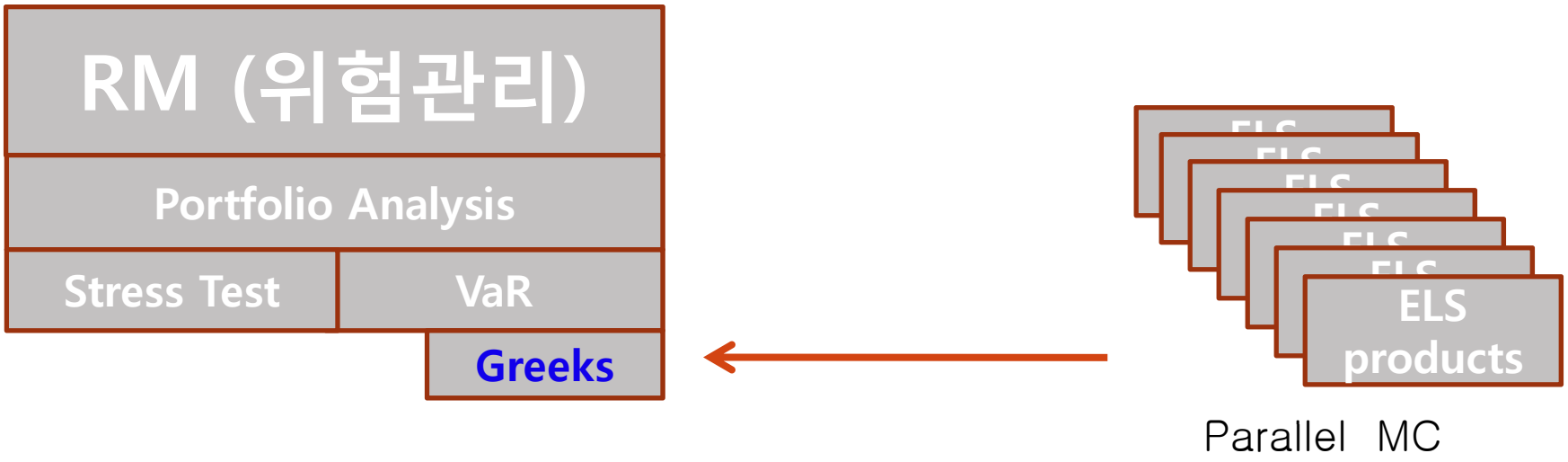
신규 상품 발생시 Front 입장



속도는 single FDM보다 빠르고 개발주기는 훨씬 짧음

Application for Massively Parallel MC

VaR을 통한 위험관리시



위험관리용 계산시간을
획기적으로 단축시킴

Application for Massively Parallel MC

채평사

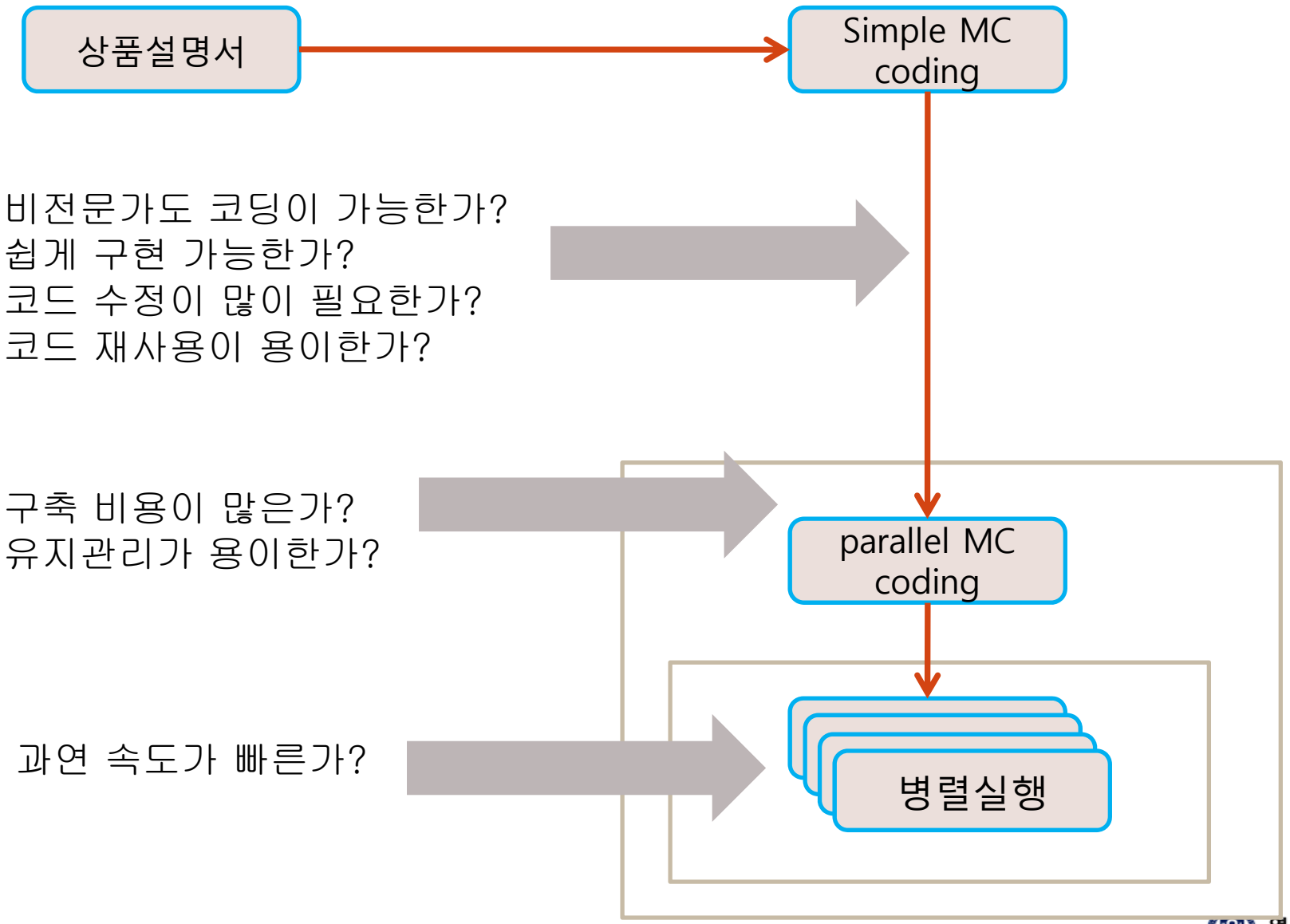
가격 baord

Parallel MC를 사용하여
기존 MC의 가속화

수많은 상품들

Parallel MC

Parallel MC구현의 용이성



병렬 프로그래밍을 위한 시스템 구축

하드웨어 구조

KISTI 수퍼컴퓨터 4호기 Tachyon

구 분	내 용
모델명	SUN Blade 6048
블레이드 노드	성능 : 24TFlops(Rpeak) 노드 : 188개(컴퓨팅)
CPU	AMD Opteron 2.0GHz 4개 노드당 : 16 Core (Quad Core) 총 : 3,008 개 (전체)
메모리	32GB(노드)
스토리지	207TB(디스크), 422TB(테이프)
노드 간 네트워크	Infiniband 4X DDR

HPC in cluster

여러대의 SMP를 네트웍으로 연결

Each Nodes has 2-8 CPUs
4~10 Gflops in DP for each CPUs
Each Nodes connected by Infiniband, Gigbit Ethernet
1024+ nodes system is possible

Support standard OpenMP & MPI library
Benefit : develop envirinment, Technical supports from vender
PRNGs library for Monte Carlo simulation

MC의 경우 1만배 이상 속도 향상을 기대할 수 있음 : 하지만 구축 및 관리비용이 많음

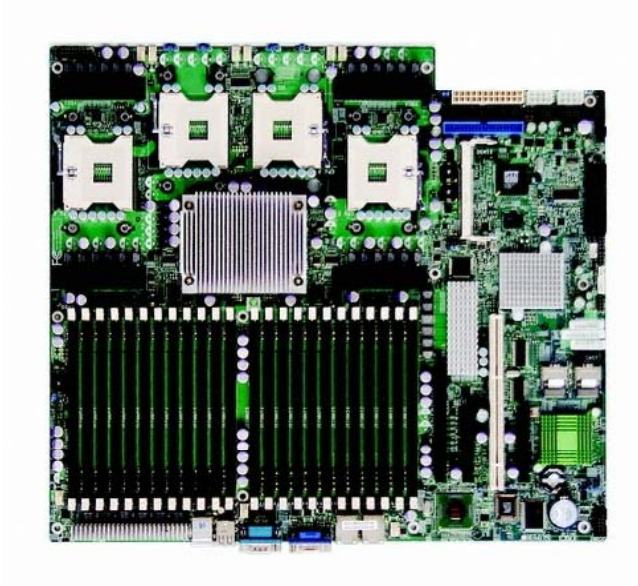
Too expensive
기상청 15Tflops 서버 도입비용 – 200억 이상 (전기, 쿨링, 공간, 네트웍, 관리자 등 유지비 필요)

금융권에 필요한 성능의 서버
HPC for 400 Gflops 서버 도입비용 - 수억원 이상 (유지비, 설치공간 등 별도) IBM, HP 등
개발용 8 workstation : 300만원~1천만원 정도면 whitebox 구축(제작)가능
개발용 2 node 2*8 core cluster : 1천만원 정도면 whitebox 구축 가능

금융권 HPC용 16코어 SMP머신

도입비용 약 2000만원 가량

HP ProLiant DL580G5 Rack Type
4U case 사용
인텔 제온 E7340 CPU 4개 : **16코어**
메모리 8GB



Node당 100Gflops 유지
Tachyon 1 노드와 유사한 성능

8노드 도입시 약 3억원 정도 예상됨

성능 : 64배 향상

국내 금융권 도입 : 비용대비 성능향상의 미약으로 HPC 도입이 잘 안됨

수퍼컴퓨터 4호기 Tachyon on KISTI

구 분	내 용
계정	SRU 100시간 : 100만원 (기업) SRU 10시간 : 10만원(학생)
접속환경	ssh 접속 30분
실행환경	CentOS 4.4 - Linux 환경
컴파일러	GCC, PGI Compiler , Intel Compiler
병렬라이브러리	OpenMPI
수치라이브러리	IMKL, IMSL 등

계정 사용을 통해 HPC 도입 효과를 미리 검토해 볼 수 있음

작업노드(프로그래밍, 디버깅)

Tachyon Server는 SSH를 통해 접속해야 하고
30분까지로 접속시간이 제한되어 있음

따라서, 다음의 시스템에서 OpenMP, MPI 코딩을 작성, 디버깅하고
최종 결과물은 Tachyon에서 실행하는 방법을 사용한다.

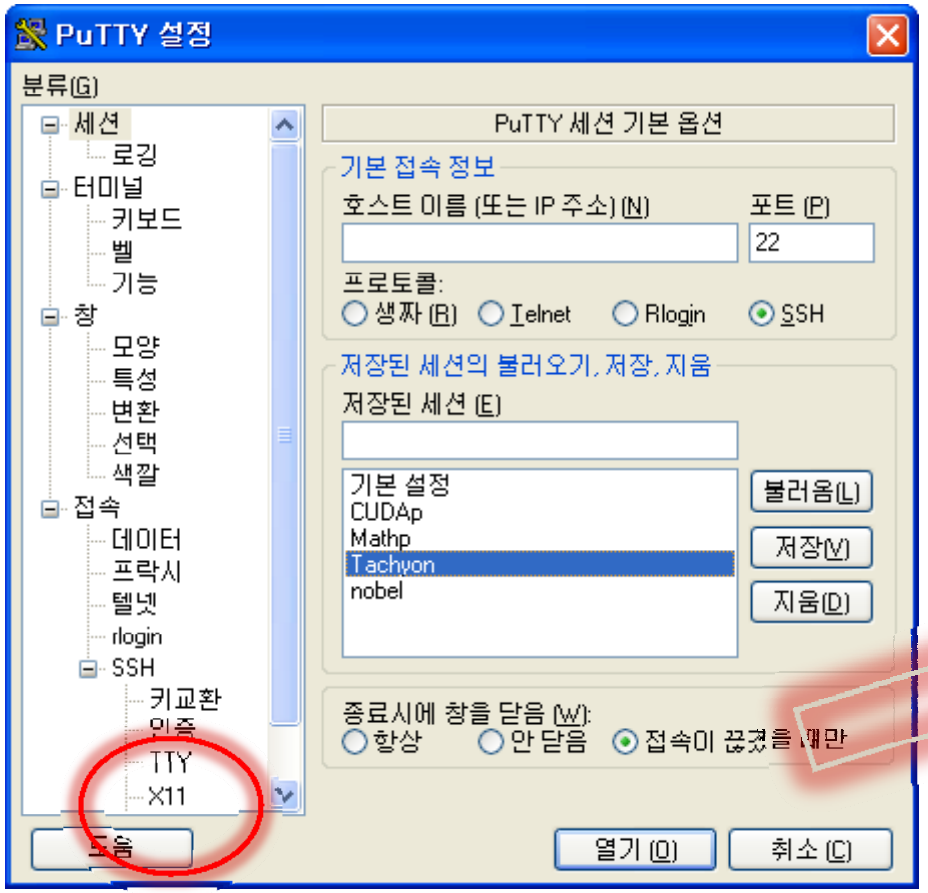
1. Windows XP 환경 (PC) : GUI 환경에서 programming
MS Visual Studio 2005+ MPICH2

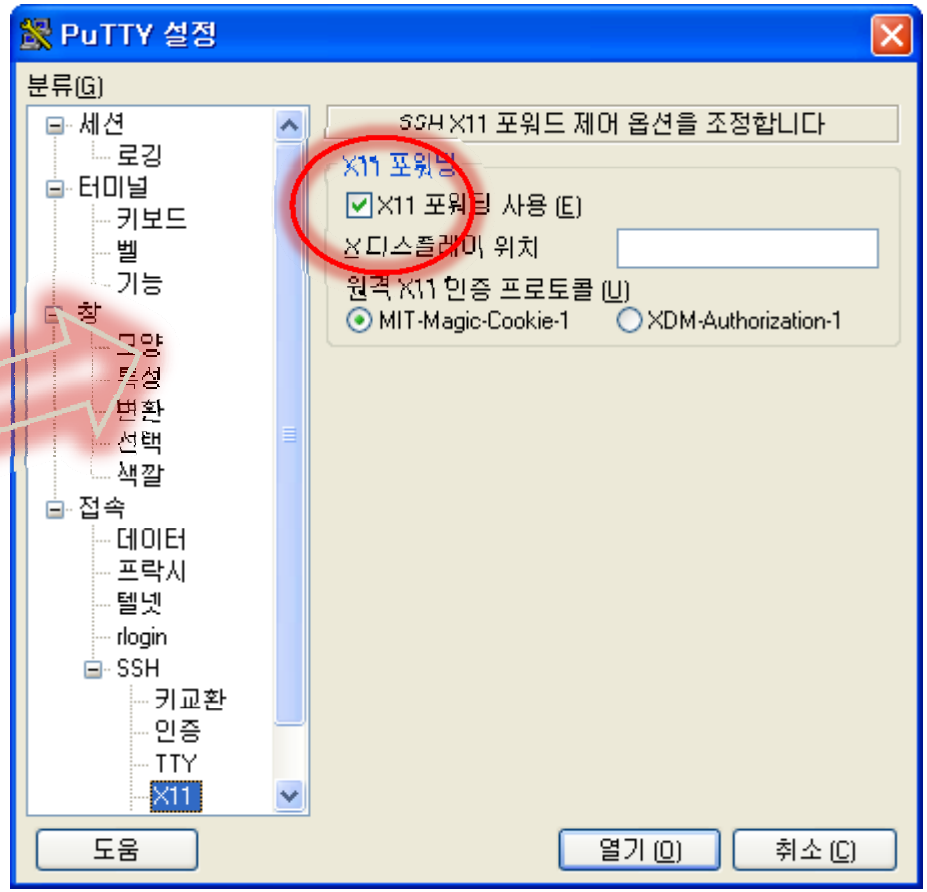
2. Linux 환경 (server) : linux에서 실행 테스트
ICC 10.1 + OpenMPI

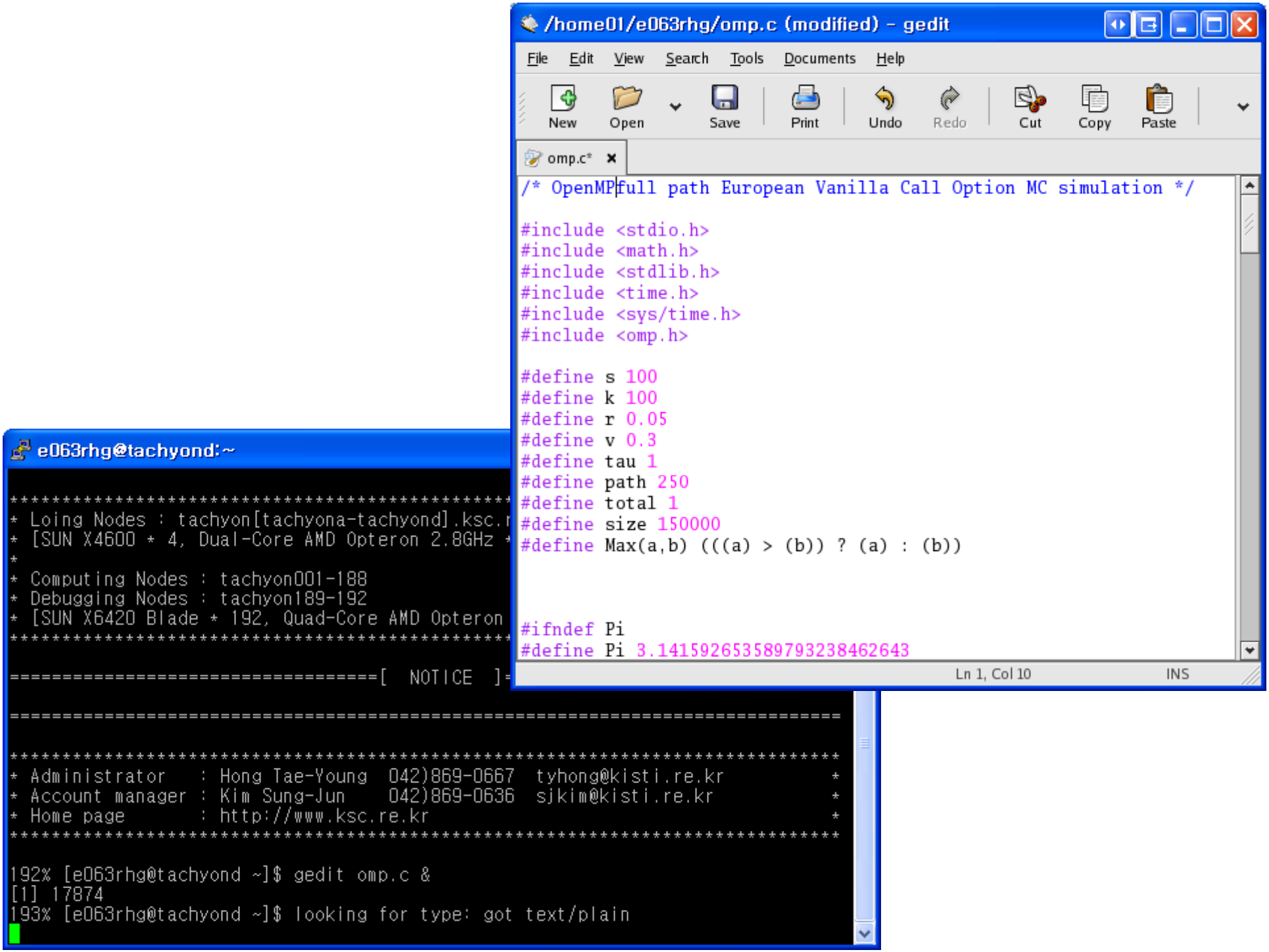
3. Tachyon Server : 최종 코드를 job_batch 실행

SSH 원격 접속환경

Windows 환경에서 PuTTY를 통해 접속






/home01/e063rhg/omp.c (modified) - gedit
File Edit View Search Tools Documents Help
New Open Save Print Undo Redo Cut Copy Paste
omp.c*
/* OpenMP full path European Vanilla Call Option MC simulation */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#define s 100
#define k 100
#define r 0.05
#define v 0.3
#define tau 1
#define path 250
#define total 1
#define size 150000
#define Max(a,b) (((a) > (b)) ? (a) : (b))
#ifndef Pi
#define Pi 3.14159265358979323846264 3
Ln 1, Col 10
INS
e063rhg@tachyond:~
* Loing Nodes : tachyon[tachyona-tachyond].ksc.
* [SUN X4600 * 4, Dual-Core AMD Opteron 2.8GHz *
* Computing Nodes : tachyon001-188
* Debugging Nodes : tachyon189-192
* [SUN X6420 Blade * 192, Quad-Core AMD Opteron
[NOTICE]
* Administrator : Hong Tae-Young 042)869-0667 tyhong@kisti.re.kr *
* Account manager : Kim Sung-Jun 042)869-0636 sjkim@kisti.re.kr *
* Home page : http://www.ksc.re.kr *
192% [e063rhg@tachyond ~]$ gedit omp.c &
[1] 17874
193% [e063rhg@tachyond ~]$ looking for type: got text/plain

병렬 환경

- **SMP 머신 : OpenMP 프로그래밍**

  16 core 지원 : 2천만원 미만 (16배 성능향상 보장)

- **Cluster 머신 : MPI 프로그래밍**

  64 core 지원 : 1억원 미만 : (60배 성능향상 보장)

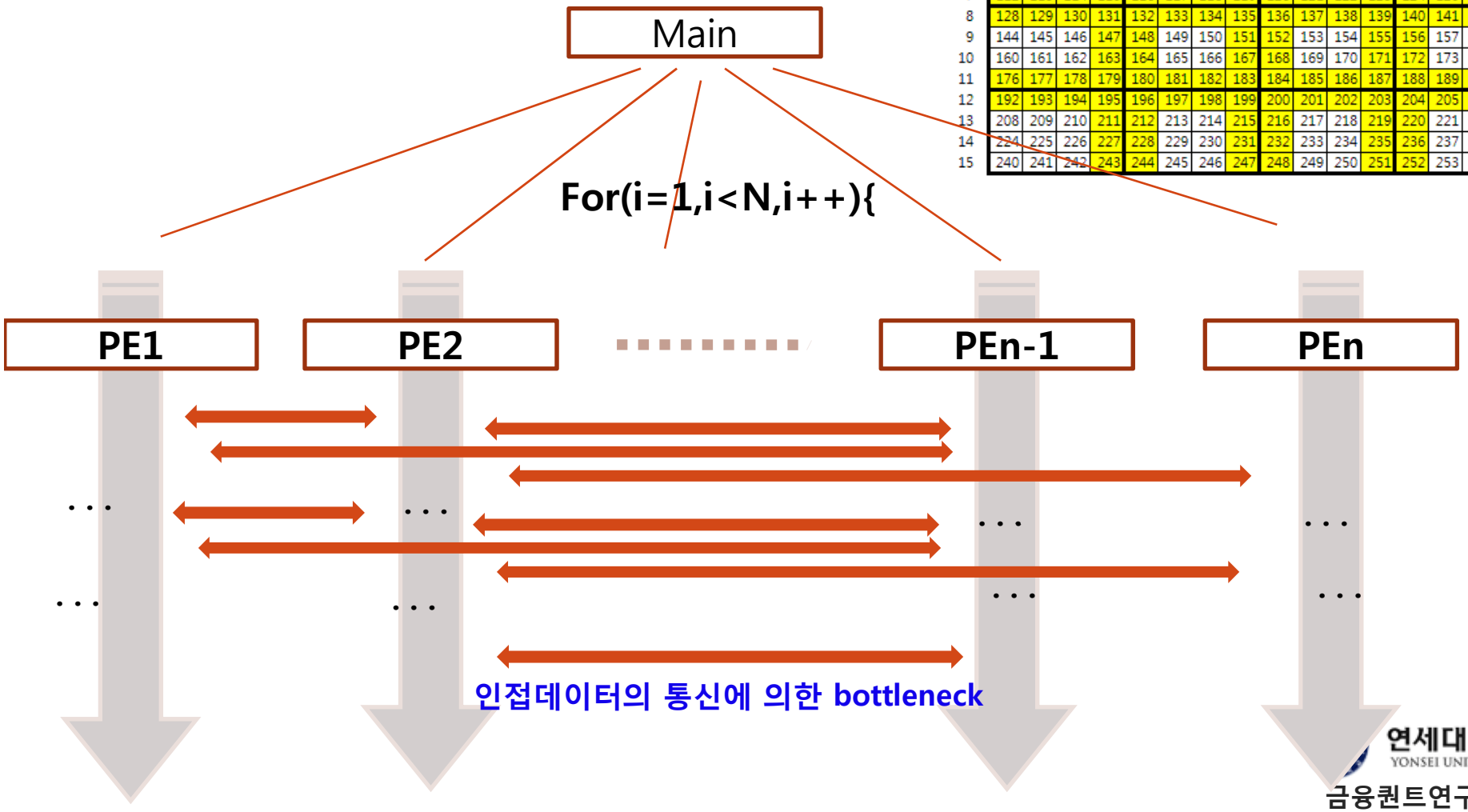
→ OpenMP, MPI : (무료)

→ IMSL, IMKL : 병렬 라이브러리 : (유료)

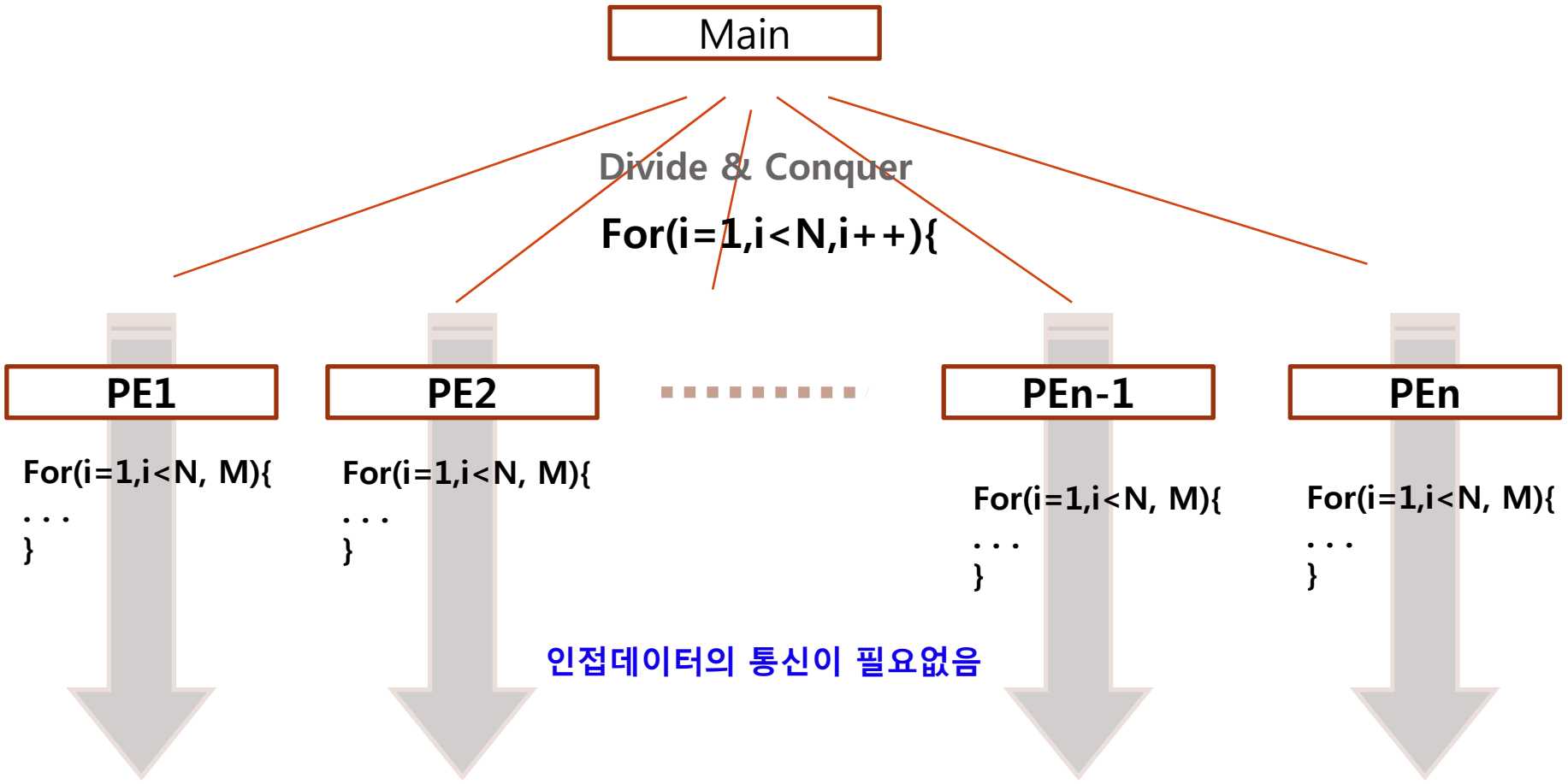
병렬 알고리즘

Parallel FDM, FEM (matrix)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
10	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
11	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
12	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
13	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
14	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
15	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255



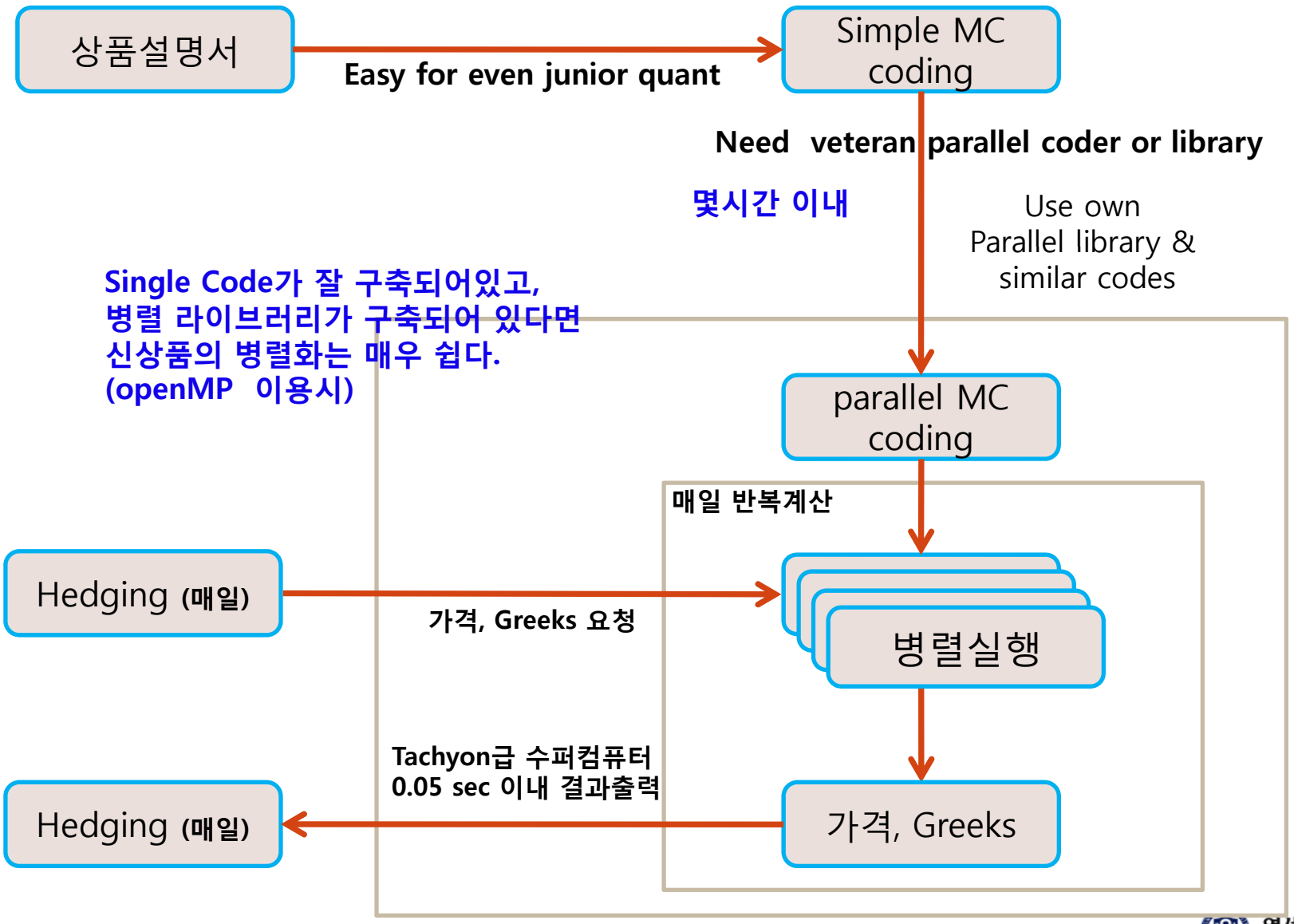
Parallel MC simulation



인접데이터의 통신이 필요없음

: MC 병렬 scailability가 좋다

Parallel Monte Carlo Simulation in Finance



LCG32

$$X_{n+1} = (a \cdot X_n + c) \bmod m$$

	M	a	c
Visual C/C++	2^{32}	214013	2531011
GNU C	2^{32}	69069	5
Unix (LCG48)	2^{48}	25214903917	11
ANSI C	2^{32}	1103515245	12345
Numerical Recipes	2^{32}	1664525	1013904223

Period : $2^{32}-1 = 4294967295 = 4.2 * 10^9$ 약 42억개

Parallel RNGs

Split Method

```
for (i=0; i<N-1; i++ )
```

```
for (i=istart; i<iend; i++ )
```

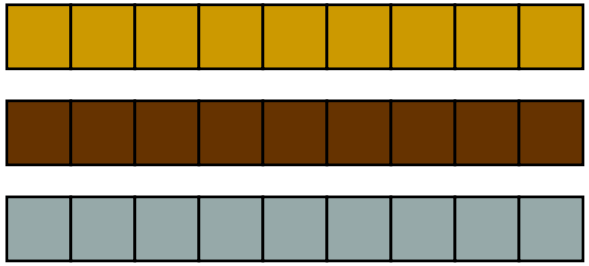
CPU ID 0 CPU ID 1 CPU ID 2

Multiseed Method

```
for (i=0; i<N-1; i++ )
```

```
srand48p(123+(1+cpuID)*45);
```

```
for (i=0; i<CHUNKSIZE-1; )
```

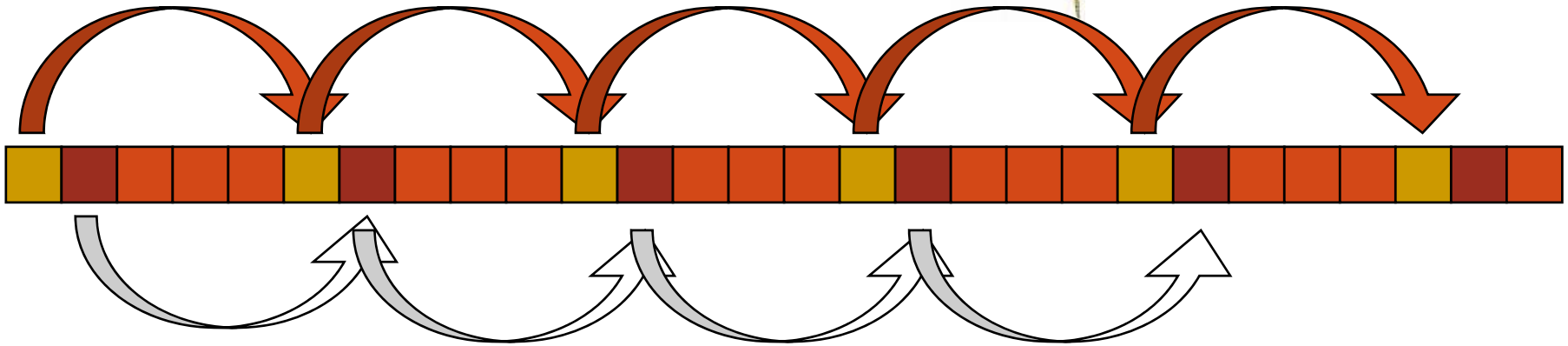


Leap Frog Method

for (i=0; i<N-1; i++)

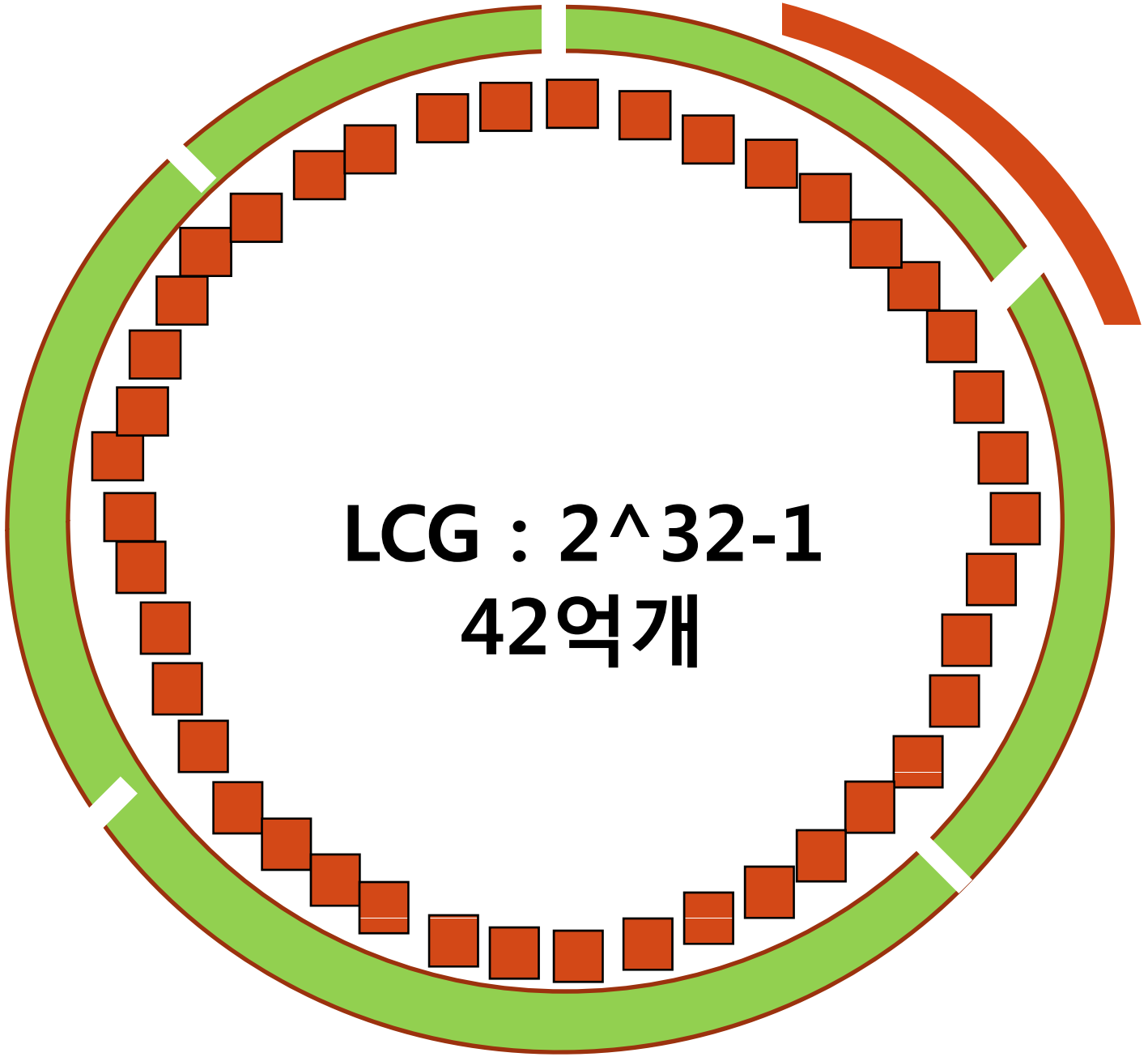
for (i=istart; i<N-CHUNKSIZE-1; CHUNKSIZE)

CPU ID 0



CPU ID 1

Problems in Parallel MC



Suitability for Parallel Method

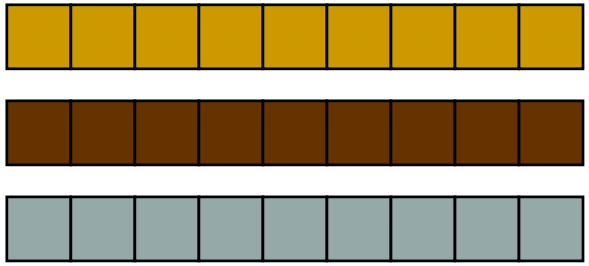
	Multiseed	Leapfrog	Splitting	D.C	J.Ah.
LCG	OK	OK	OK	X	-
RAND48	OK	OK	OK	X	-
MCG	OK	OK	OK	X	-
Lagged Pibonachi	OK	OK	OK	X	-
MWC	OK	OK	OK	X	-
MT19937	OK	X	X	OK	OK
Well RNG	OK	?	?		OK

Dynamic Creation

for (i=0; i<N-1; i++)

Complex dividing for non-overlapping cycle

for (i=0; i<CHUNKSIZE-1;)

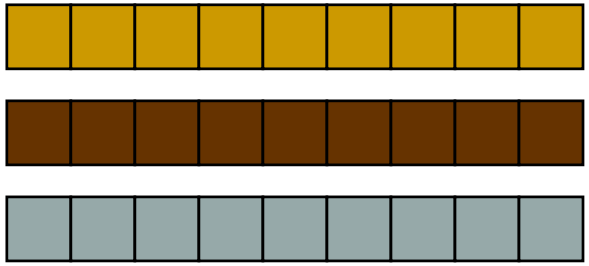


Jump-Ahead Method

for (i=0; i<N-1; i++)

`Divide sequence with GF(2) polynomial`

for (i=0; i<CHUNKSIZE-1;)



CPU ID 0

CPU ID 1

CPU ID 2

Algorithm for MT19937

Step 0. $\mathbf{u} \leftarrow \underbrace{1\cdots1}_{w-r}\underbrace{0\cdots0}_{r}$;(bitmask for upper $w-r$ bits)

$\mathrm{ll} \leftarrow \underbrace{0\cdots0}_{w-r}\underbrace{1\cdots1}_{r}$;(bitmask for lower r bits)

$\mathbf{a} \leftarrow a_{w-1}a_{w-2}\cdots a_1a_0$;(the last row of the matrix A)

Step 1. $i \leftarrow 0$

$\mathbf{x}[0], \mathbf{x}[1], \cdots, \mathbf{x}[n-1] \leftarrow$ "any non-zero initial values"

Step 2. $\mathbf{y} \leftarrow (\mathbf{x}[i]$ AND $\mathbf{u})$ OR $(\mathbf{x}[(i+1) \bmod n]$ AND $\mathrm{ll})$;(computing $(\mathbf{x}_i^u|\mathbf{x}_{i+1}^l)$)

Step 3. $\mathbf{x}[i] \leftarrow \mathbf{x}[(i+m) \bmod n]$ XOR $(\mathbf{y} >> 1)$

$$\text{XOR} \begin{cases} 0 & \text{if the least significant bit of } \mathbf{y} = 0 \\ \mathbf{a} & \text{if the least significant bit of } \mathbf{y} = 1 \end{cases}$$;(multiplying A)

Step 4. ;(calculate $\mathbf{x}[i]T$)

$\mathbf{y} \leftarrow \mathbf{x}[i]$

$\mathbf{y} \leftarrow \mathbf{y}$ XOR $(\mathbf{y} >> u)$;(shiftright $\mathbf{y}$ by u bits and add to $\mathbf{y}$)

$\mathbf{y} \leftarrow \mathbf{y}$ XOR $((\mathbf{y} << s)$ AND $\mathbf{b})$

$\mathbf{y} \leftarrow \mathbf{y}$ XOR $((\mathbf{y} << t)$ AND $\mathbf{c})$

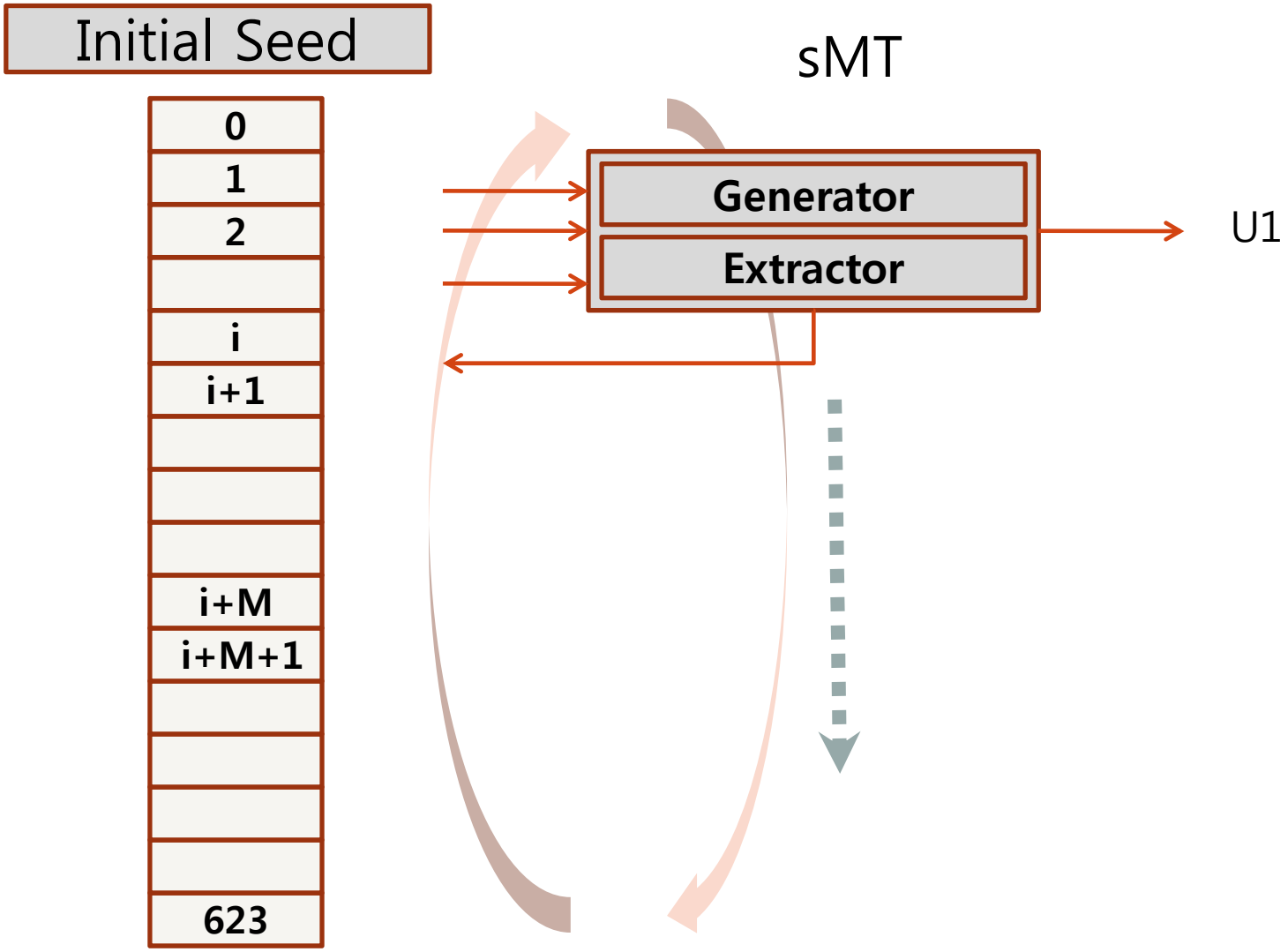
$\mathbf{y} \leftarrow \mathbf{y}$ XOR $(\mathbf{y} >> l)$

output $\mathbf{y}$

Step 5. $i \leftarrow (i+1) \bmod n$

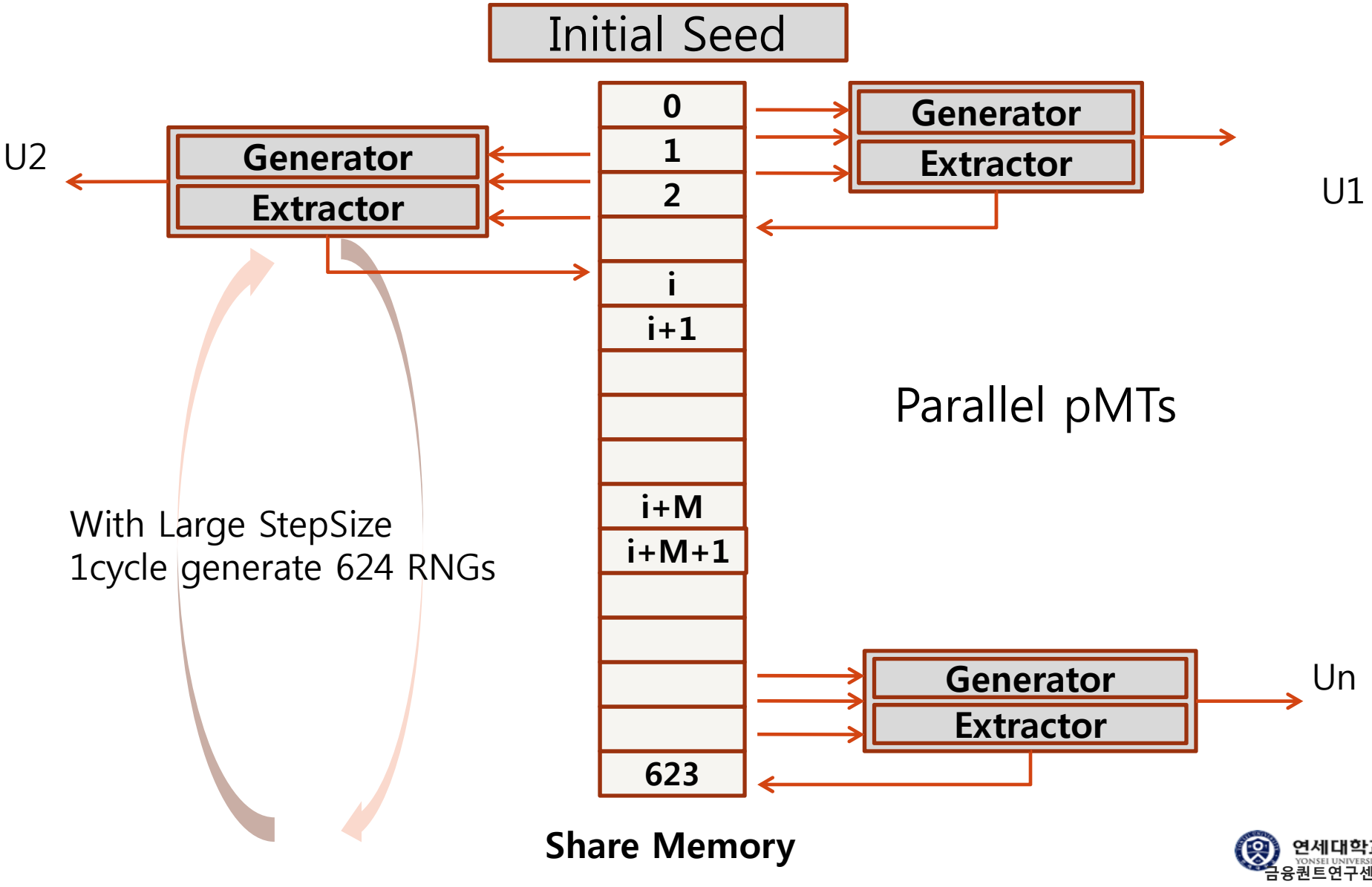
Step 6. Goto Step 2.

sMT architecture of MT19937



MT19937 병렬화

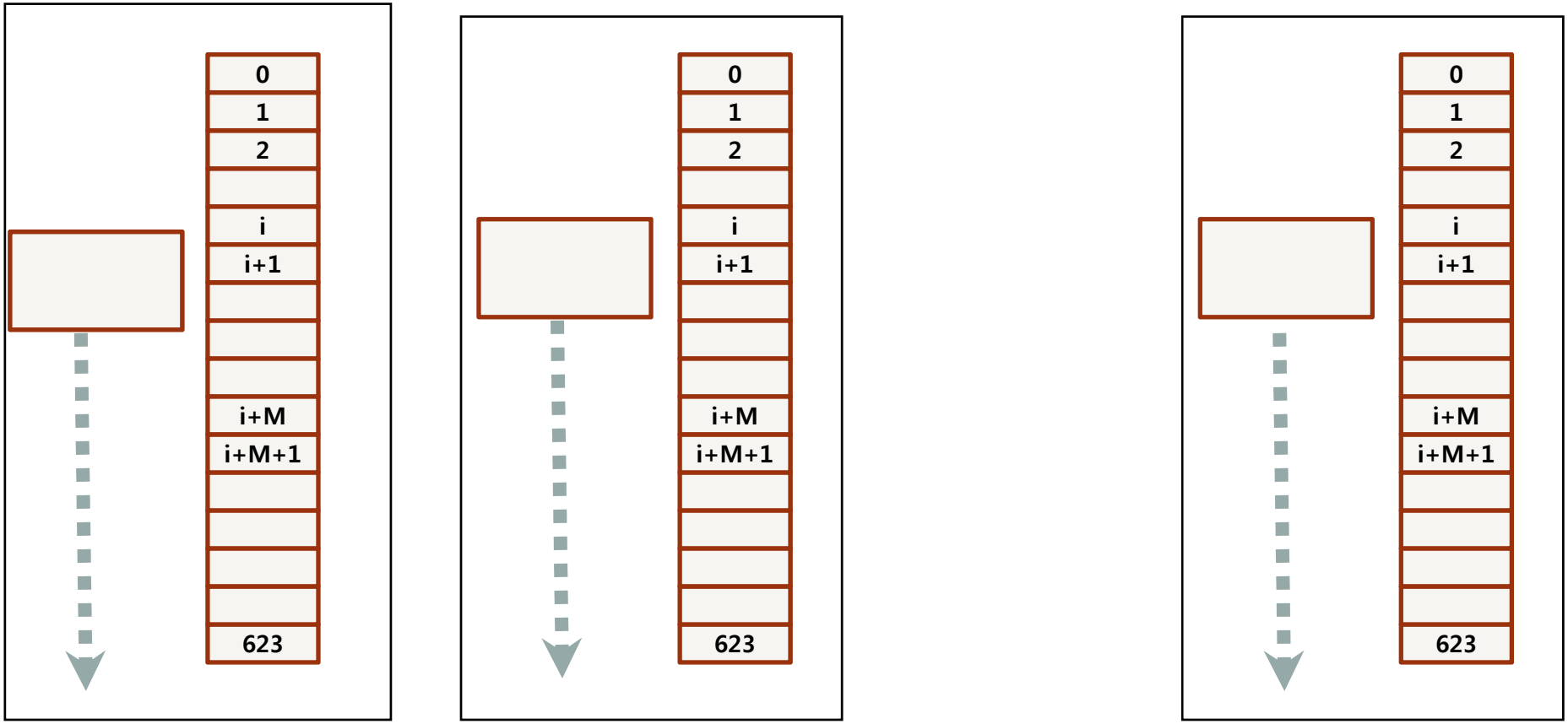
Shared pMT architecture of MT19937



Parallel sMT MT19937

Each Cores execute sMT independently

We will use this model



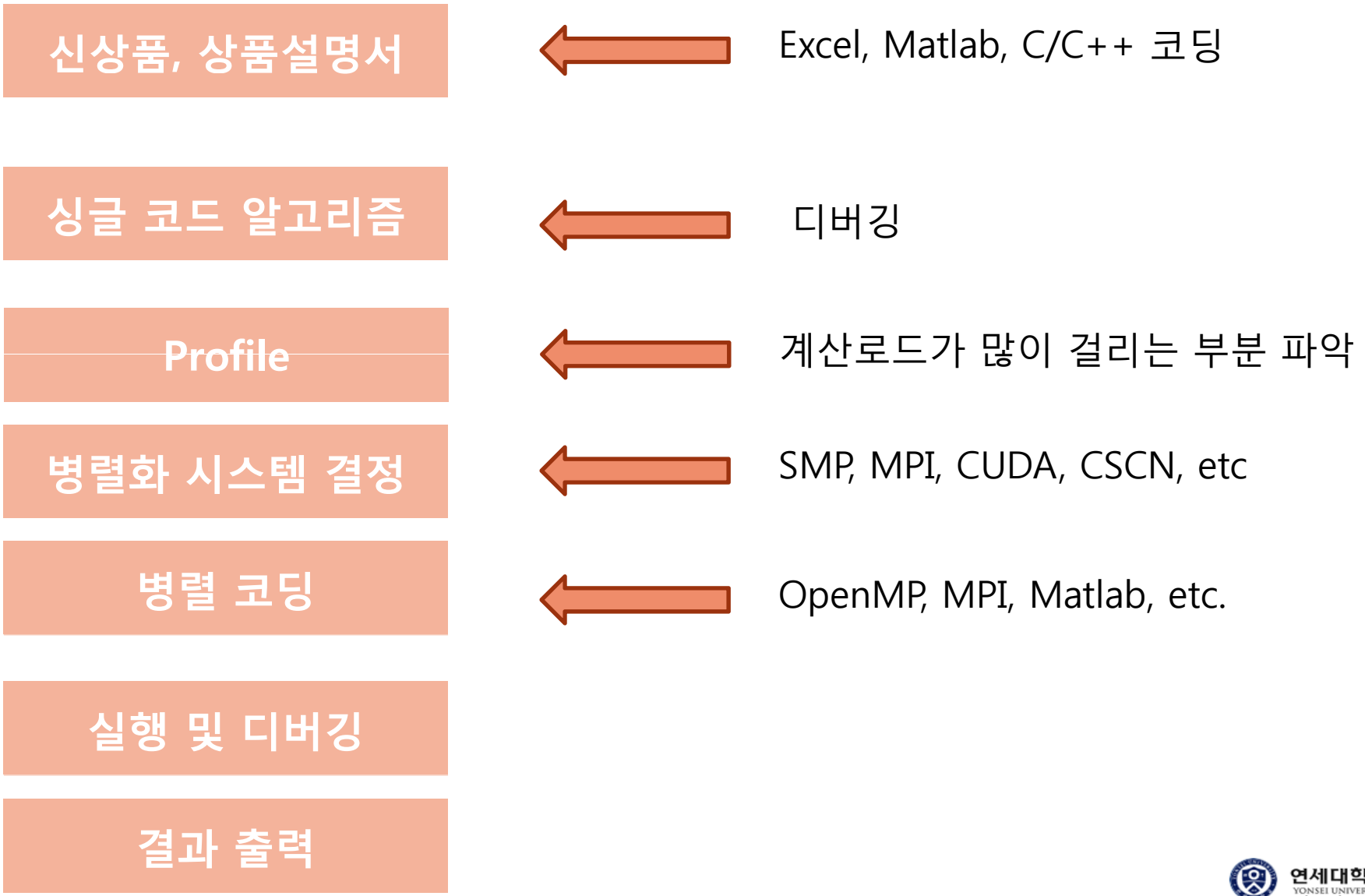
Parallel Quasi RNG

For Loop Nsim simulation
For Loop for T simulation of path

→Method 1 base p에 의한 병렬화
Split Halton with Nsim /16 -- 16 is # of core
Core has own Base **3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59**
Scramble each Sequence with T times

→Method 2 : 준주기성을 고려한 n 병렬화
select large prime such as 23
Compute Split K= Nsim/16 * 24
n=(K*coreID + i)
Parallely Compute Halton(23,n)
Scramble each Sequence with T times

실제 병렬화


신상품, 상품설명서
싱글 코드 알고리즘
Profile
병렬화 시스템 결정
병렬 코딩
실행 및 디버깅
결과 출력
Excel, Matlab, C/C++ 코딩
디버깅
계산로드가 많이 걸리는 부분 파악
SMP, MPI, CUDA, CSCN, etc
OpenMP, MPI, Matlab, etc.

금융문제의 병렬화 기법

사용

방법1
총 M회 시뮬레이션을 병렬처리
각 N개의 Core가 M/N 회 시뮬레이션 실행
각 Core가 독립적으로 Boxmuller 및 RNG 생성
→ 1/N으로 계산 시간 단축
→ 장점 : single code의 90% 이상 그대로 사용

방법2
1. RNG를 병렬화하여 미리 생성
2. N개의 시뮬레이션을 병렬처리
3. N개의 시뮬레이션 평균

→ 조기상환시 RNG 생성 시간 낭비

방법3 Table Technique
RNG를 병렬화하여 미리 생성(1회)
→ memory에 load하여 필요시 사용

각 상품별로 (K회)
M/N회의 시뮬레이션을 병렬처리

→ 많은 상품을 계산해야하는
시스템에서 적합
1/K*N 으로 계산시간 단축
충분한 메모리가 확보 및 분산처리
Memory access bottleneck 발생

모듈화 작업시
single code를 그대로 사용가능

Single Code

병렬 몬테카를로 시뮬레이션 예제

1. **Full-path 유럽형 Vanilla Call 옵션** : reference & benchmark 용
S : 100 , K : 100 , r : 0.03 , v : 0.3, T : 1

2. **ELS pricing** : 삼성증권 1909호 주식연계증권

3년 만기
2 star : POSCO 일반주, S-Oil

12 chance : 디지털 옵션

Double Barrier
Down barrier : 장중체크 (둘 중 하나라도 하락한계)
Up barrier : 매일 종가 체크(두개다 상승한계)
Step-down : 없음, 동일한 조기상환 조건 (Digital Option 형태)

지급 : 조기상환시 +2 영업일

4. 수퍼컴퓨터를 이용한 parallel Monte Carlo Simulation I

OpenMP를 이용한 병렬화

OpenMP 개념

메모리를 공유한 SMP 머신 (core2 duo 등의 형태)에서 병렬코딩함.
16코어 머신을 이용하면 MC의 경우 약 16배의 속도 향상 기대

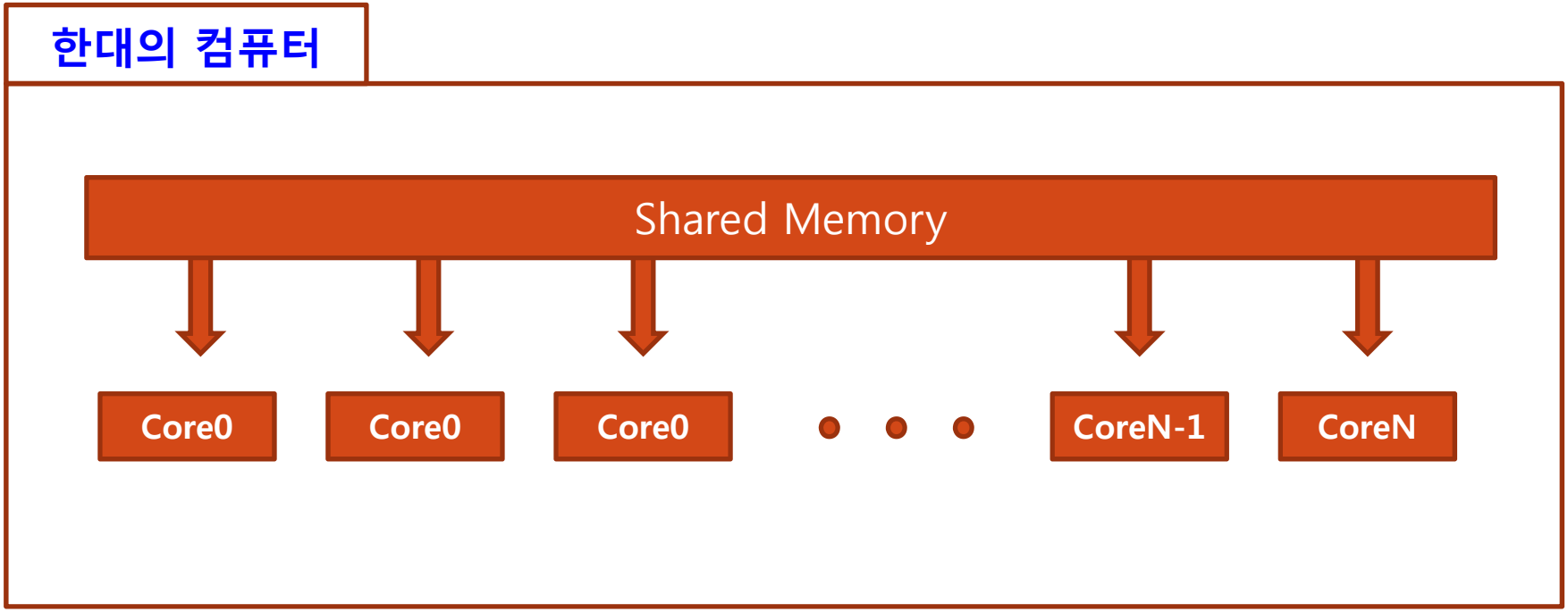
C/C++ 언어와 Fortran 언어를 지원함

Windows 환경은 Visual Studio 2003 이상에서 기본 지원
Unix 환경의 경우 최신 버전 기본 지원
Linux의 경우 GCC 2.4.2 이상에서 지원

OpenMP 명령어를 통해 컴파일러가 자동으로 병렬화

For Loop를 쉽게 병렬화 할 수 있고,
각 Core가 메모리를 공유하기 때문에 병렬 코딩이 매우 편함

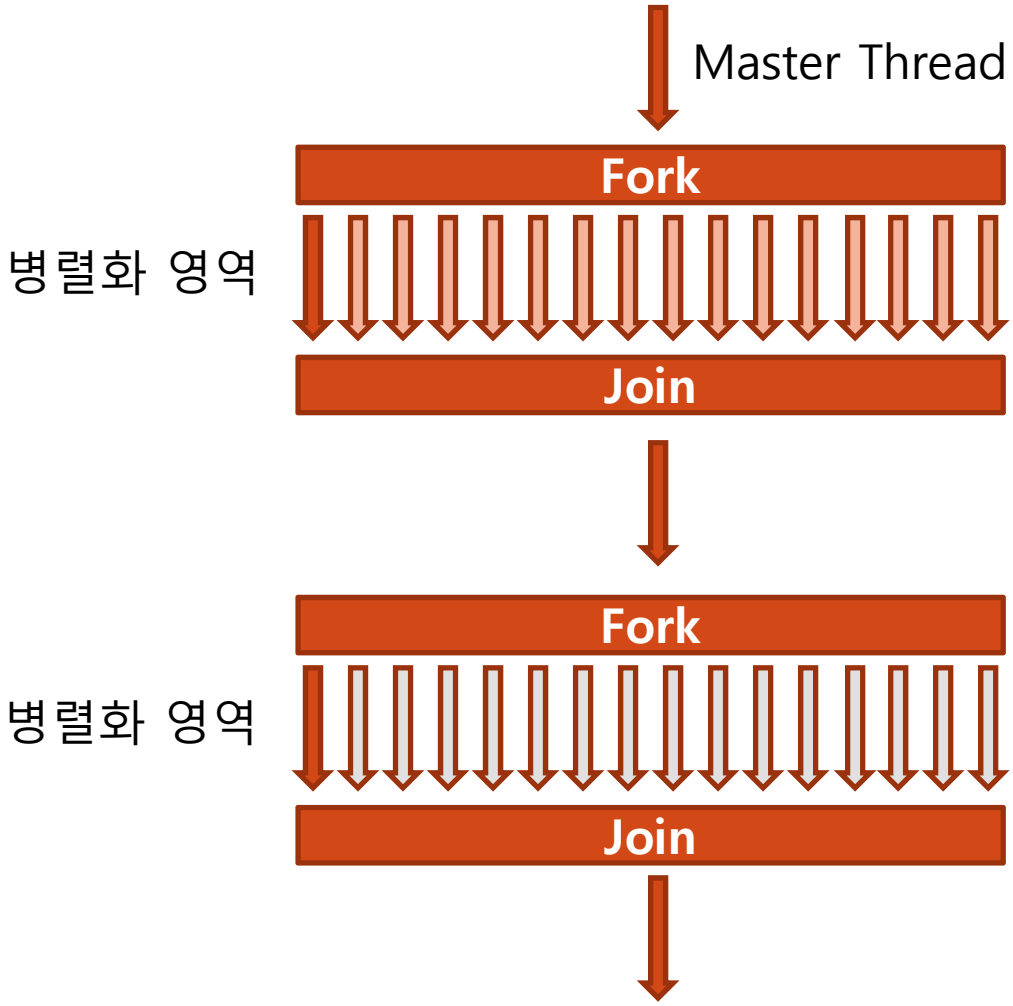
SMP



각 코어가 모두 하나의 메인보드 위에 있기 때문에 모두 같은 메모리를 사용
특별히, 통신 코딩을 할 필요가 없음

➔ OpenMP 코딩의 편리함

OpenMP 개념



OpenMP 병렬화 예약어들

omp_set_num_threads(16);　　총 16개의 thread 사용설정

totalN=omp_get_num_threads;　전체 병렬화 개수 파악

tid=omp_get_thread_num();　각 병렬 프로세서 번호 인식

#pragma omp　지시어

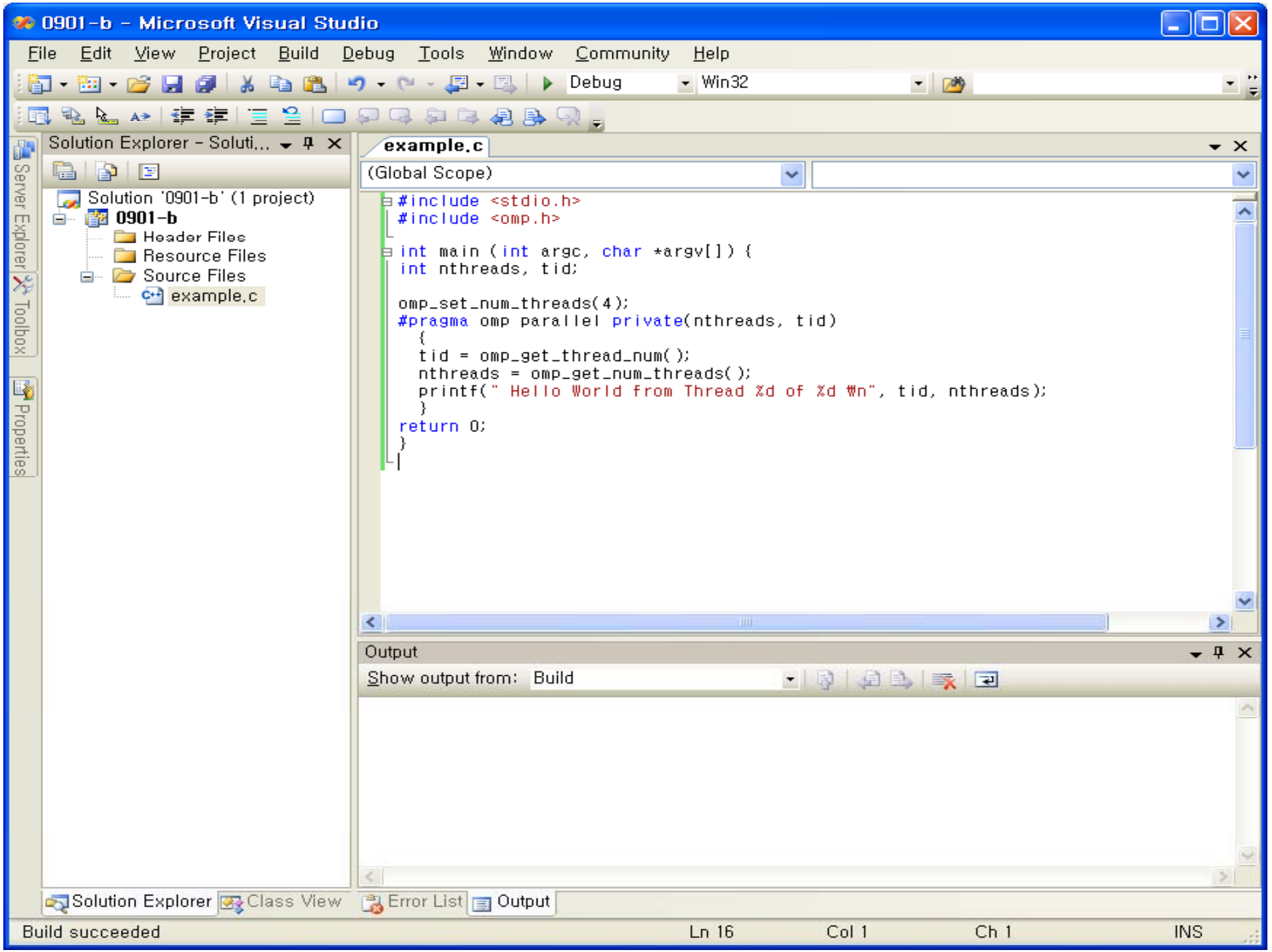
#ifdef_OPENMP　순차 프로그래밍에서도 사용가능하도록 프로그래밍

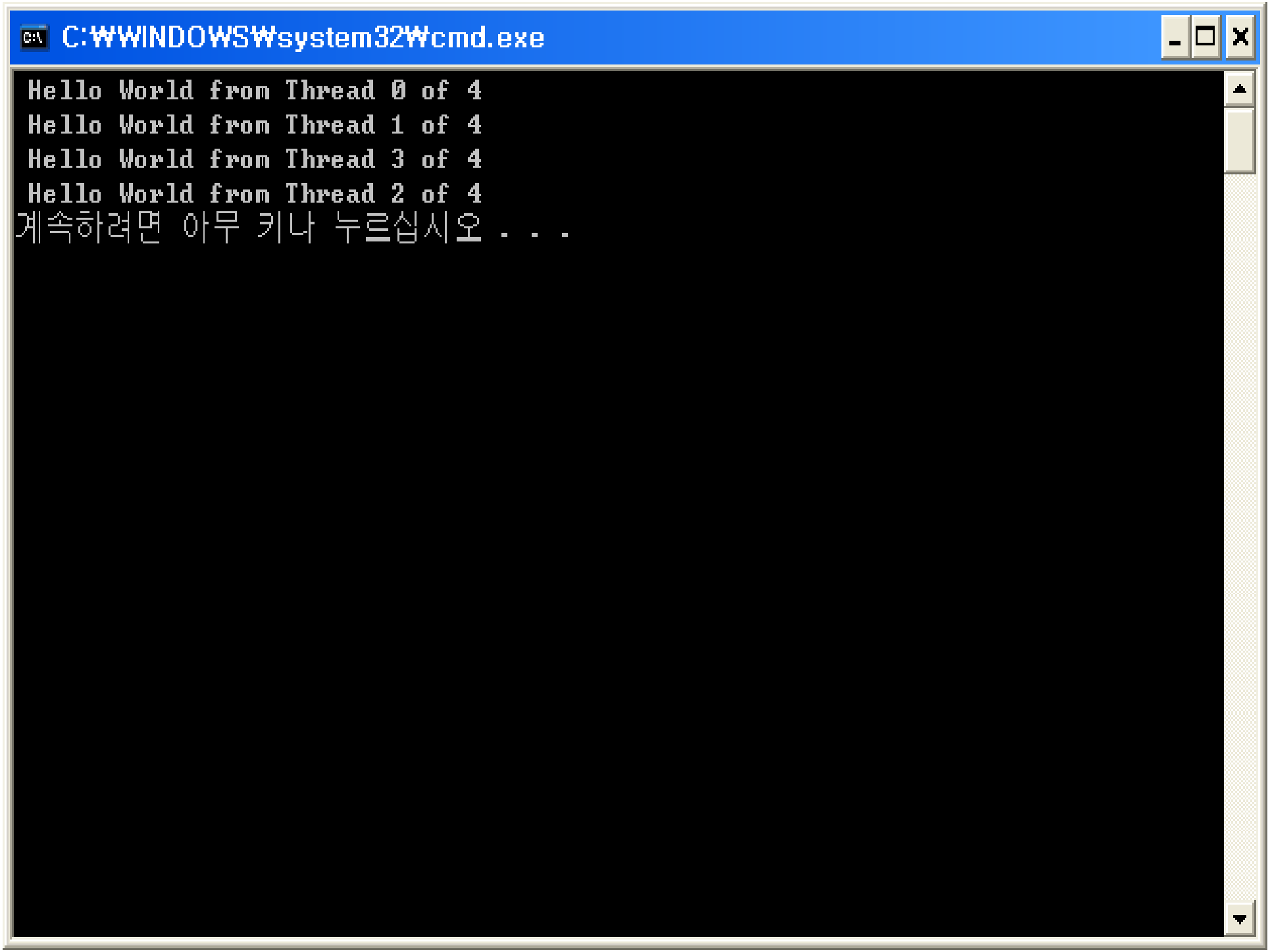
parallel for critical
private() shared() schedule()

OpenMP 예제1

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
int nthreads, tid;

omp_set_num_threads(4);

#pragma omp parallel private(nthreads, tid)
  {
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf(" Hello World from Thread %d of %d ₩n", tid, nthreads);
  }
return 0;
}
```

병렬화 영역


0901-b - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
Debug
Win32
Solution Explorer - Soluti...
Solution '0901-b' (1 project)
0901-b
Header Files
Resource Files
Source Files
example.c
Server Explorer
Toolbox
Properties
example.c
(Global Scope)
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
int nthreads, tid;
omp_set_num_threads(4);
#pragma omp parallel private(nthreads, tid)
{
tid = omp_get_thread_num();
nthreads = omp_get_num_threads();
printf(" Hello World from Thread %d of %d \n", tid, nthreads);
}
return 0;
}
Output
Show output from: Build
Solution Explorer
Class View
Error List
Output
Build succeeded
Ln 16
Col 1
Ch 1
INS


C:₩WINDOWS₩system32₩cmd.exe
Hello World from Thread 0 of 4
Hello World from Thread 1 of 4
Hello World from Thread 3 of 4
Hello World from Thread 2 of 4
계속하려면 아무 키나 누르십시오 . . .

OpenMP 병렬화 방법

```
for(i=0; i<m; i++)
        {
        a[i] = b[i*n]*c[0];
        for(j=1; j<n; j++)
            a[i] += b[i*n+j]*c[j];
        }
```


자동 병렬화

```
#include <omp.h>

#pragma opm parallel for shared(m,n)  private(i,j)
    for(i=0; i<m; i++)
        {
        a[i] = b[i*n]*c[0];
        for(j=1; j<n; j++)
            a[i] += b[i*n+j]*c[j];
        }
```

```
#pragma omp parallel shared(fsum, N)  private(tid,fsum_local,NNN)
{
   NNN = 0;
   fsum_local = 0;
   tid = omp_get_thread_num();
   printf("%d \t",omp_get_num_threads());

#pragma omp for
   for ( i=0; i< Nsim ;i++ )
   {
      xt1=s;
      xt2=s;
      for(m=0; m<path; m++)
      {
         xx1 = myrand()/(RAND_MAX+1.0);if(xx1==0.0) xx1=0.0000000000001;
         xx2 = myrand()/(RAND_MAX+1.0);if(xx2==0.0) xx2=0.0000000000001;
         normal1=sqrt(-2.0*log(xx1 ))*cos(2.0*3.14159265358979323846*xx2 );
         xt1= xt1 + r * xt1 * dt + v*xt1*sqrt(dt)*normal1;
      }
      oprice=Max(xt1-k,0);
      fsum_local =fsum_local+ oprice;
      //printf("%f",op);
   }
#pragma omp barrier
#pragma omp critical (update_sum)
   {
      fsum +=fsum_local;
      stop=clock();
      printf("\n%d \t %20.17f \t %20.17f \n",tid,fsum, fsum_local) ;
   }
} // end of OpenMP
results = (double) 1/Nsim *  exp(-1* r * tau)* fsum;
stop=clock();
htime = 0.001*difftime(stop,start); //windows
printf("\n%19.17f \t %19.17f     %7.5f \n",results,results-bsp,  htime) ;
```

RNG 병렬화 코딩 필요

전체 병렬화 영역

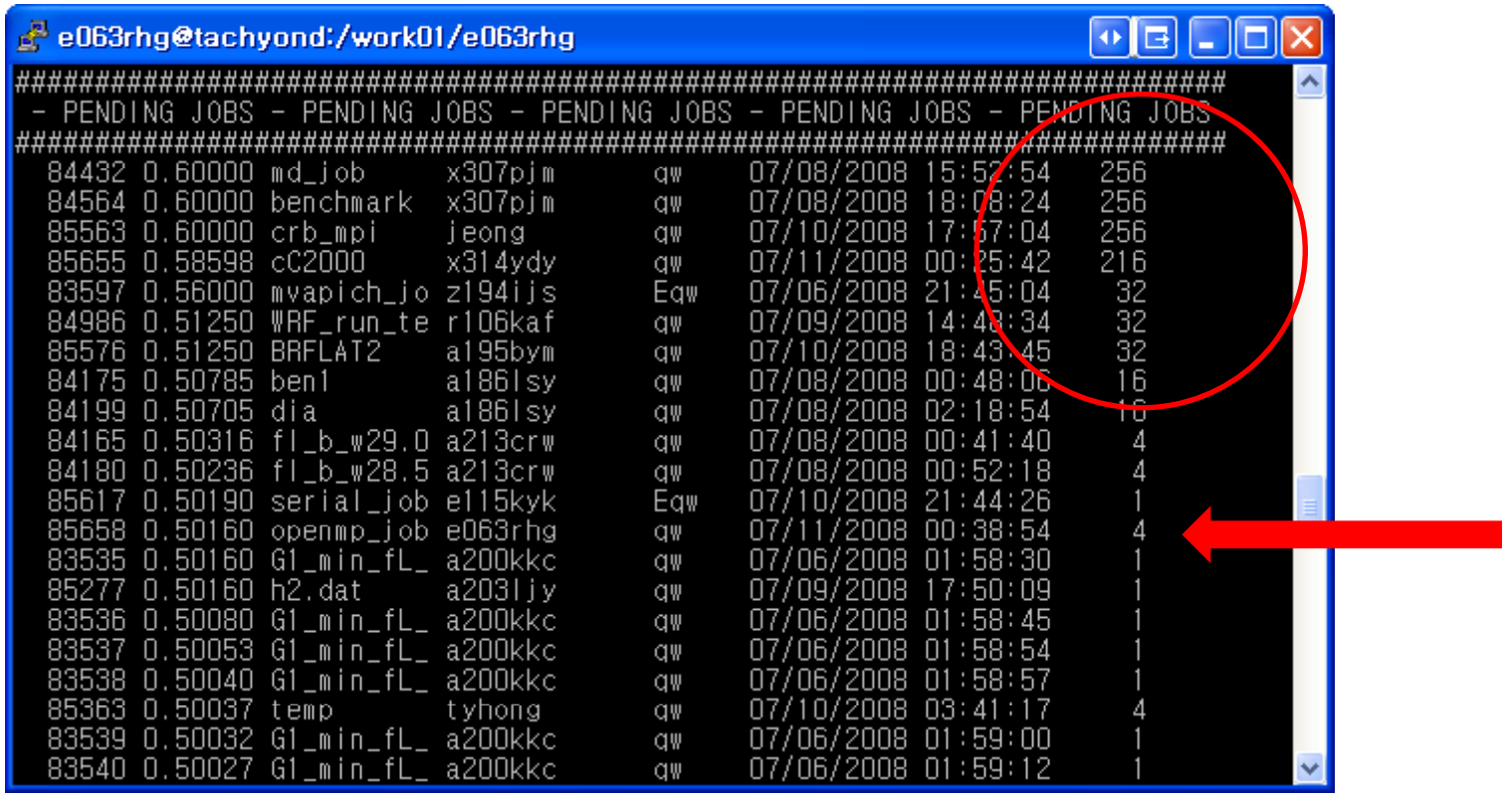
Loop 병렬화 부분

결과 Reduction 부분

수퍼컴퓨터 Tachyon OpenMP job batch scheduler

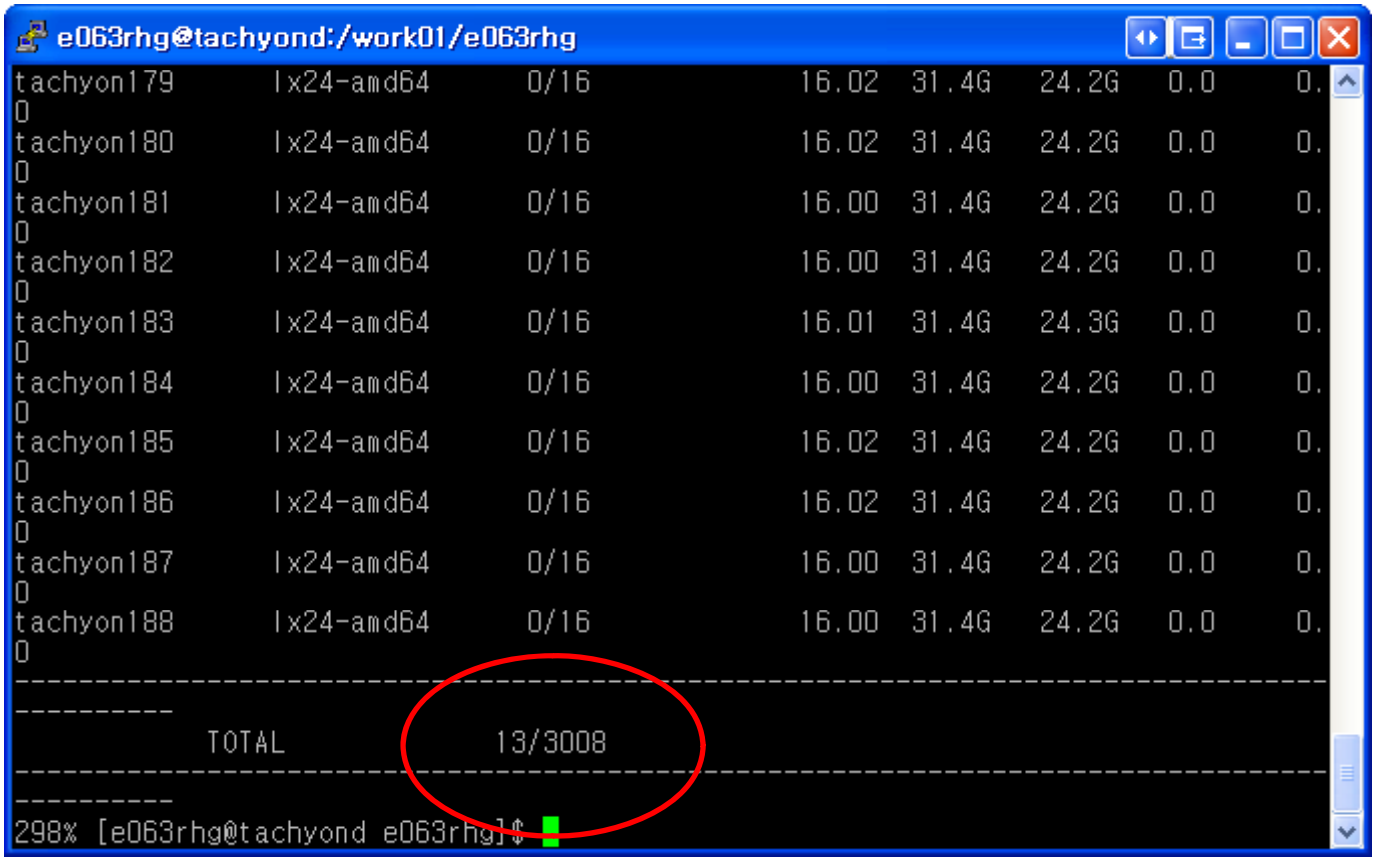
```
282% [e063rhg@tachyond e063rhg]$ vi a.sh
#!/bin/bash
#$ -V
#$ -cwd
#$ -N openmp_job
#$ -pe openmp 4
#$ -q small
#$ -R yes
#$ -wd /work01/e063rhg/
#$ -l h_rt=00:01:00
#$ -M myEmailAddress
#$ -m e
export OMP_NUM_THREADS=4
/work02/e063rhg/omp.exe >result.txt
```

Job scheduler에서 대기중인 사람들



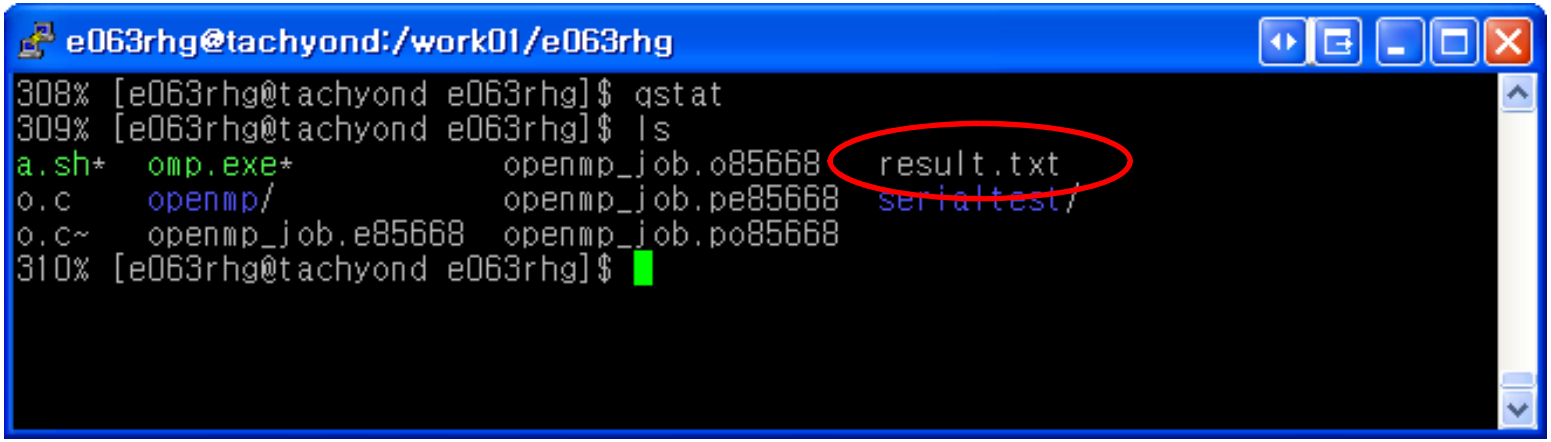
예상실행시간 40 sec의 job을 약 40분 기다려야함
Scheduler 조정을 통해 제일 적게 기다리는 편

Tachyon idle core 현황



6월 30일부터 현재까지 계속 job이 waiting 상태로,
수퍼컴퓨터 사용의 의미가 없음 OpenMP, MPI test 불가

Job Schedule 실행된 모습




```
================================
Black-Scholes : 14.23124493415723357
1
3     -1073749904      566413.67638386972248554     566413.67638386972248554

3     -1073749904      1132306.36511257803067565    565892.68872870830819011

3     -1073749904      1701503.34236495988443494    569196.97725238185375929

3     -1073749904      2254231.80765507556498051    552728.46529011556413025

14.29527750057961200      0.06403256642237842
elapsed time is 44.892820
```

OpenMP 결과

1. 수퍼컴퓨터에서는 1~16배까지 core 개수를 늘리면 scailability가 증가

2. 유휴 CPU가 부족하여 scailability를 테스트 하기 어려웠음.

3. CUDAp, MATHp, Windows PC 등에서
   1~4 core 기반의 scailability 쉽게 체크 가능

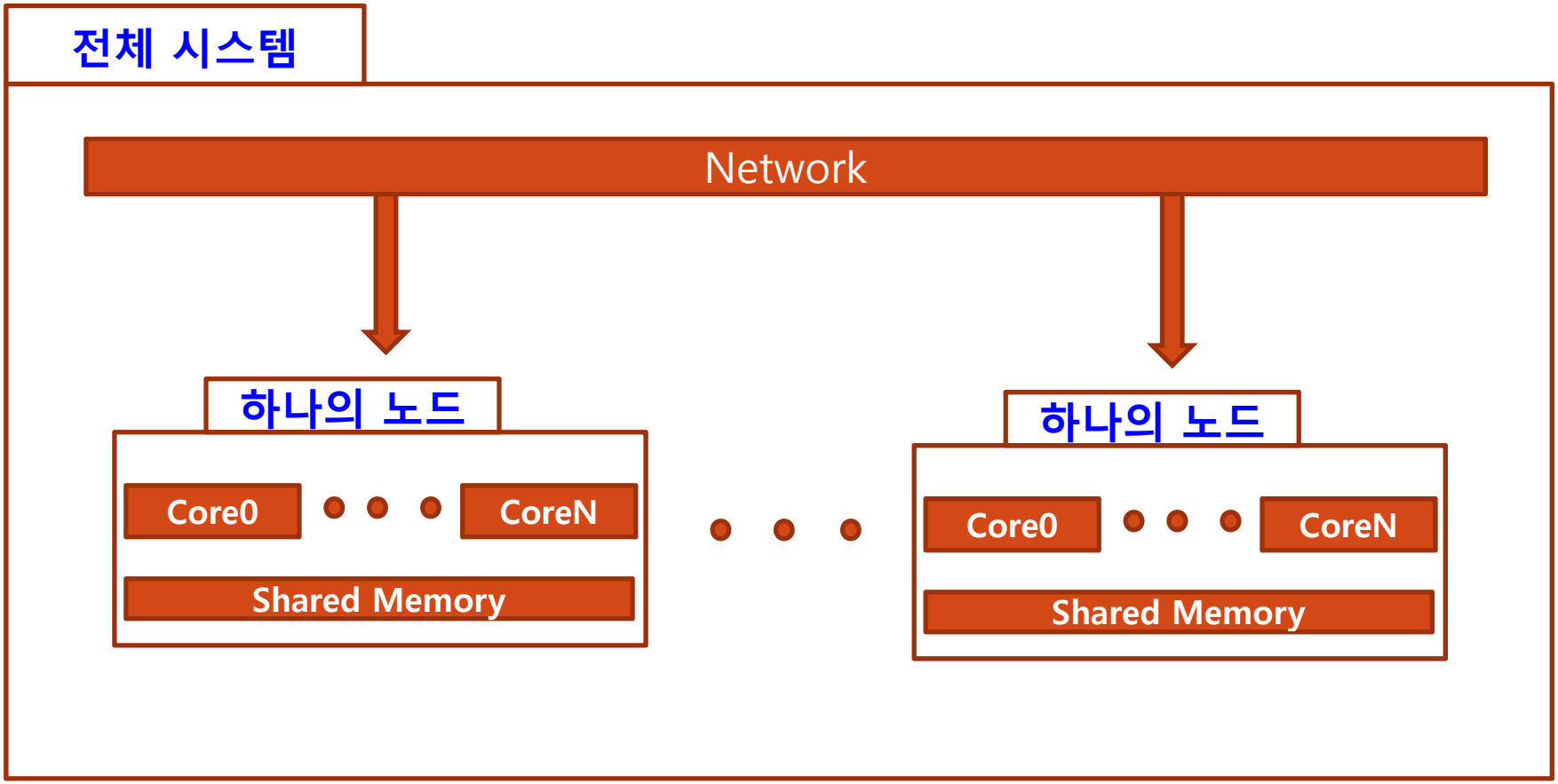
4. 간단히 rand()함수를 이용한 loop 병렬화 시 오히려 속도 감소

   ➔ memory bank conflict, shared static variable 문제

5. 수퍼컴퓨터를 이용한 병렬 Monte Carlo Simulation II

MPI 병렬화

Cluster



각 노드에서 각각 프로그램을 실행해줘야함 (mpirun)
각 노드의 메모리는 서로 공유하지 않으므로 서로 통신해 줘야함

➔ MPI 코딩의 복잡함

MPI 개념

MPIRUN
MPI로 병렬 프로그래밍된 실행 명령을 각각의 node에 복사하여
각각의 노드가 병렬 명령을 실행할 수 있도록 지원함

MPI setting에 각각의 node 설정을 해주어야함 : 관리의 복잡성 발생

각 Node는 네트웍을 통해 연결되어 있으므로, 서로 메모리를 공유하지 않음.
각 메모리는 서로 독립적으로 작동하므로 PDE solving, Matrix 병렬화 등에서는
서로의 메모리 정보를 서로 통신을 통해 업데이트 해줘야함

MPI는 병렬화 보다는 오히려 통신 프로그래밍을 쉽게 해준 역할을 함.
→OpenMP와는 다르게 프로그래머가 병렬화 작업, 메모리 공유를 직접 해줘야 함

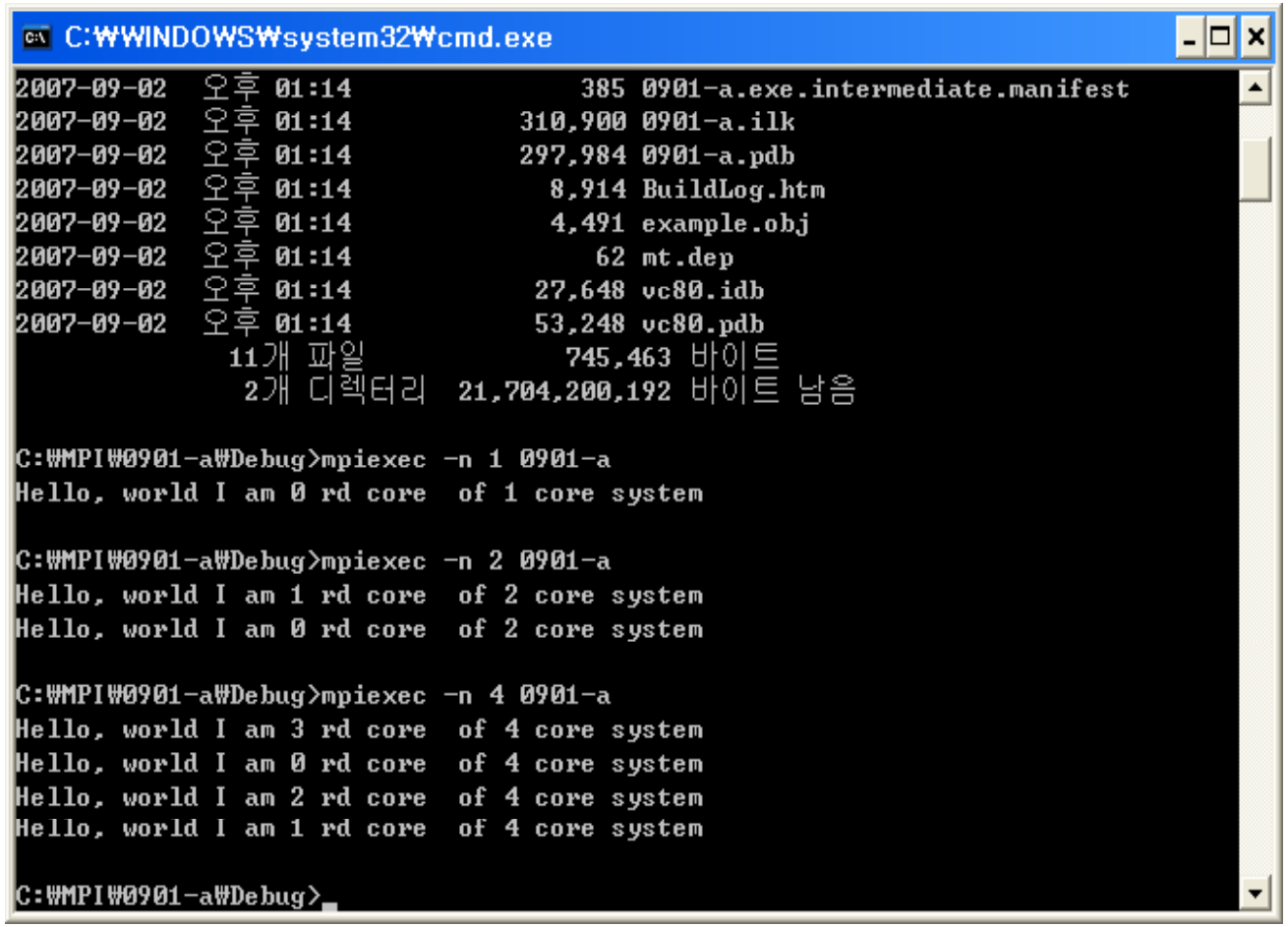
MPIEXEC, POE 등의 MPI Launcher를 통해 실행해줘야한다.
전용서버의 경우 scheduler 대신 plink를 활용하면 유용하다.

MPI 예제1

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[]) {
        int rank, size;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);

        printf("Hello, world I am %d rd core  of %d core system₩n",rank,size);

        MPI_Finalize();
        return 0;
}
```

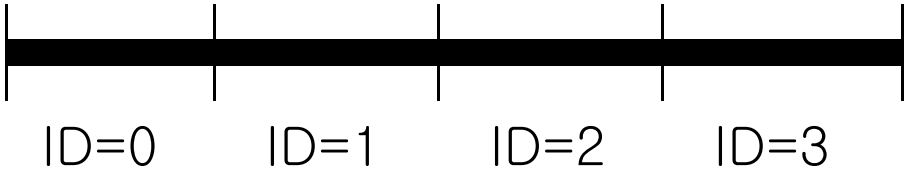
MPIexec

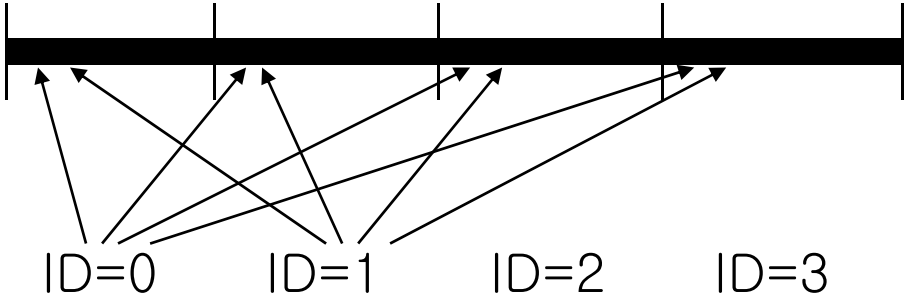
다음은 mpiexec를 통해 core를 1개, 2개 4개로 확장시켰을 때의 실행 결과를 나타내고 있다.



Job 분할 방법

How do we divide job (FOR LOOP) ?





Loop 분할 방법들

단순분할

1

```
for (i = 0; i < Nsim/N; i++) {
        ….
   }
```

Block 분할

2

```
i_start = Tid  * (Nsim /N);
i_end = i_start + (Nsim /N);
if (Tid == (N-1))  i_end = N;

for (i = i_start; i < i_end; i++) {
        ….
   }
```

순환분할

3

```
for (i = Tid; i < Nsim; i+= N) {
        ….
}
```

블록-순환분할

4

```
for (i = n1*block*myrank;
        i < n2; i+=nprocs*block) {

   for (j = jid; j < min(ij+block-1); i+=n2N) {
        ….
}
```

para_range(n1,n2,n3,n4,n5,n6) 함수

Para_range 함수는 크게 3가지의 병렬화 방법 중 Method 2의 기법으로
For Loop를 균등 분할함

```
void para_range(int lowest, int highest,
                int nprocs, int myrank,
                int *start, int *end) {
    int wk1, wk2;

    wk1 = (highest - lowest + 1) / nprocs;
    wk2 = (highest - lowest + 1) % nprocs;
    *start = myrank * wk1 + lowest + ( (rank<wk2) ? myrank : wk2);
    *end = *start + wk1 - 1;
    if(wk2 > rank) *end = *end + 1;

}
```

$$\sum_{i=1}^{N} a(i) = \sum_{i=1}^{n_1} a(i) + \sum_{i=n_2+1}^{n_2} a(i) + \cdots + \sum_{i=n_{k-1}+1}^{N} a(i)$$

Core 0 Core 1 Core k-1

MPI 예제2 (MPI 통신)

```
#include <mpi.h>
#include <stdio.h>
#define n 100000

void para_range(int, int, int, int, int*, int*);
int min(int, int);

void main (int argc, char *argv[]){
    int i, nprocs, myrank ;
    int ista, iend;
    double a[n], sum, tmp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    para_range(1, n, nprocs, myrank, &ista, &iend);

    for(i = ista-1; i<iend; i++) a[i] = i+1;
        sum = 0.0;
        for(i = ista-1; i<iend; i++) sum = sum + a[i];
            MPI_Reduce(&sum, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
              sum = tmp;
              if(myrank == 0) printf("sum = %f ₩n" , sum);

MPI_Finalize();
}
```

MPI 예제3 Pi계산

```
#include <math.h>
#define n 100000
main(){
    int i,istep,itotal[10],itemp;
    double r, seed, pi, x, y, angle;
    pi = 3.1415926;
    for(i=0;i<10;i++) itotal[i]=0;
    seed = 0.5; srand(seed);
    for(i=0; i<n; i++){
       x = 0.0; y = 0.0;
       for(istep=0;istep<10;istep++){
          r = (double)rand();
          angle = 2.0*pi*r/32768.0;
          x = x + cos(angle);
          y = y + sin(angle);
       }
    itemp = sqrt(x*x + y*y);
    itotal[itemp]=itotal[itemp]+1;
    }

    for(i=0; i<10; i++){
        printf(" %d :", i);
        printf("total=%d₩n",itotal[i]);
    }
}
```

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#define n 100000

void para_range(int, int, int, int, int*, int*);
int min(int, int);

main (int argc, char *argv[]){
    int i, istep, itotal[10], iitotal[10], itemp;
    int ista, iend, nprocs, myrank;
    double r, seed, pi, x, y, angle;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    para_range(0, n-1, nprocs, myrank, &ista, &iend);

    pi = 3.1415926;
    for(i=0; i<10; i++) itotal[i] = 0;
       seed = 0.5 + myrank; srand(seed);
       for(i=ista; i<=iend; i++){
          x = 0.0; y = 0.0;
          for(istep=0; istep<10; istep++){
             r = (double)rand();
             angle = 2.0*pi*r/32768.0;
             x = x + cos(angle); y = y + sin(angle);
          }
          itemp = sqrt(x*x + y*y);
          itotal[itemp] = itotal[itemp] + 1;
      }

    MPI_Reduce(itotal, iitotal, 10, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    for(i=0; i<10; i++){
        printf(" %d :", i);
        printf(" total = %d₩n",iitotal[i]);
    }
MPI_Finalize();
}
```

MPI예제4 (Full Path 유렵형 Vanilla Call옵션)

```
for ( i=0; i< Nsim ;i++ )
   {
      xt1=s;
      xt2=s;
      for(m=0; m<path; m++)
      {
         xx1 = rand()/(RAND_MAX+1.0);if(xx1==0.0) xx1=0.0000000000001;
         xx2 = rand()/(RAND_MAX+1.0);if(xx2==0.0) xx2=0.0000000000001;
         normal1=sqrt(-2.0*log(xx1 ))*cos(2.0*3.14159265358979323846*xx2 );
         xt1= xt1 + r * xt1 * dt + v*xt1*sqrt(dt)*normal1;
      }
      oprice=Max(xt1-k,0);
      fsum =fsum+ oprice;
}
results = (double) 1/Nsim *  exp(-1* r * tau)* fsum;
stop=clock();
htime = 0.001*difftime(stop,start); //windows
printf("\n%19.17f \t %19.17f     %7.5f \n",results,results-bsp,  htime) ;
```

Loop분할필요

병렬 RNG필요
multiseed method

MPI reduce 필요

MPI화 코드

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#define n 150000
void para_range(int, int, int, int, int*, int*);
int min(int, int);

main (int argc, char *argv[]){
    int i,  m;
    int ista, iend, nprocs, myrank;
    double r, seed, pi, x, y, angle;
    Nsim = n;
    금융관련 변수 설정은 모두 생략함 (single 코드와 동일)

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    para_range(0, Nsim, nprocs, myrank, &ista, &iend);
        seed = 0.5 + myrank; srand(seed);
        for ( i=ista; i< iend ;i++ )
        {
            xt1=s;
            xt2=s;
            for(m=0; m<path; m++)
            {
                xx1 = rand()/(RAND_MAX+1.0);if(xx1==0.0) xx1=0.0000000000001;
                xx2 = rand()/(RAND_MAX+1.0);if(xx2==0.0) xx2=0.0000000000001;
                normal1=sqrt(-2.0*log(xx1 ))*cos(2.0*3.14159265358979323846*xx2 );
                xt1= xt1 + r * xt1 * dt + v*xt1*sqrt(dt)*normal1;
            }
            oprice=Max(xt1-k,0);
            fsum_local =fsum_local+ oprice;
        }
    MPI_Reduce(&fsum_local, &fsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Finalize();
 results = (double) 1/Nsim *  exp(-1* r * tau)* fsum;
 printf(" Option Price : %23.17f " results);
}
```

병렬화 블록

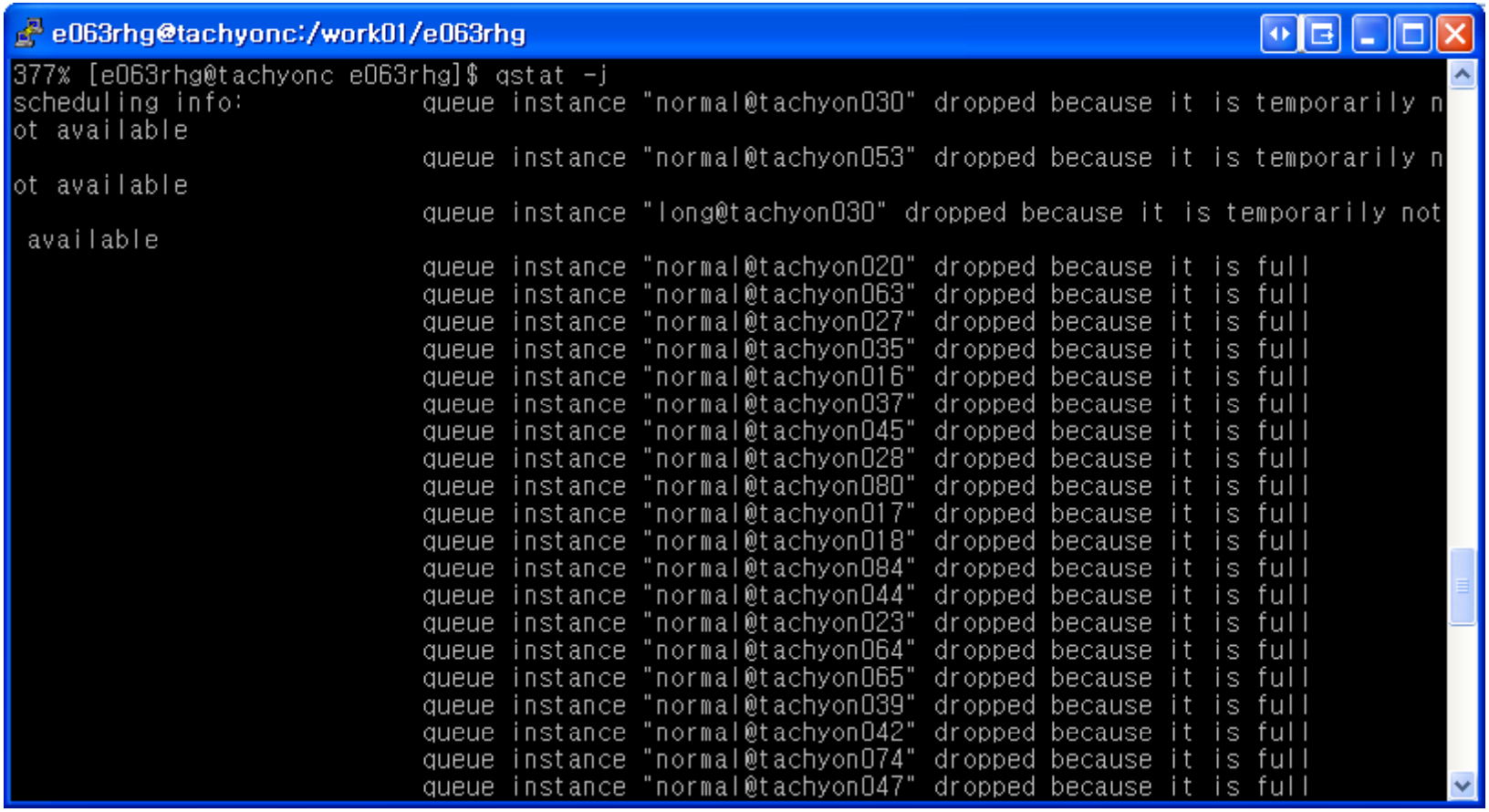
Multiseed방법사용

Single code 그대로 사용

Core간 데이터 통신

상황에 따라
추가적 RNG코딩

Tachyon : all node is full



MPI 결과

수퍼컴퓨터 센터의 tachyon, nobel 서버 모두 busy

➔ 연세대학교 수학과 PDE팀 4 node 8 core server
➔ kisti Hamel Cluster (5년 이상 된 Cluster로 성능이 많이 떨어지만 사용자 적음) 에서 테스트

Linear Scailability : CPU 개수와 속도향상의 선형성이 보장됨
(단, 병렬 RNG 코딩을 해줘야함.)

2. Monte Carlo Simulation은 병렬화의 Scailability가 매우 좋은 편임
N개의 Core 사용시 1/N으로 계산시간 단축효과, 1/sqrt(N)의 정확도 향상

현재의 정확도를 유지하면서 속도를 향상시킴 : 1/N

수퍼컴퓨터를 이용한 몬테카를로 병렬화

1. 계산 전용 서버는 자체 구축 필요 (KISTI의 경우 Job schedule : 최소 30분)

2. MPI의 경우 직접 병렬코딩을 하고, MPIRUN을 실행해 줘야 하지만
잘 만들어진 single 코드가 존재하는 경우 몬테카를로 병렬화는 어렵지 않다.

3. 병렬화를 통한 속도 향상
200초 걸리는 시뮬레이션 문제의 경우

2노드	32 core system 구축시 6.3초	0.5억원
8노드	128 core system 구축시 1.6초	2억원 +알파
16노드	256 core system 구축시 0.8초 예상됨	4억원 +알파
32노드	512 core system 구축시 0.4초 예상됨	8억원 +알파

➔ 32노드의 사용시 MC의 편리함과 FDM 수준의 속도를 얻을 수 있음

Parallle Quasi MC를 사용할 경우 8노드 정도로 FDM 수준의 속도향상 예상됨

대안적 방법

대안적 방법

PC용 CPU는 계산위주이 아닌 범용으로 개발됨

계산위주란?
빠른 처리속도와 더불어 CPU 내부의 On-chip 메모리가 큼
여러 개의 명령을 동시에 실행, ALU, FP 연산용 모듈이 많이 내장

Intel은 내부 메모리를 줄인 저가버전 등등을 출시 (가격경쟁을 위해)

따라서, 계산에 최적화된 하드웨어를 통한 병렬화 기법이
최근 HPC 업계에 주목받고 있음.

대표적인 예가
IBM Cell BE, Clearspeed, GPGPU 등임

대안1 IBM BladeCenter QS20 , QS21, QS22

Each Nodes has 3.2Ghz Cell BE processor
One Cell BE has 8 SPE units for computing

QS20 : 204 Gflops in SP, 21 Gflops in DP
QS21 : 408 Gflops in SP, 42 Gflops in DP
QS22 : 460 Gflops in SP, 217 Gflops in DP
QS21 : 5 Tflops in SP/ 588Gflops in DP in for blade chassis

Cell BE CPU ws developed for PS3

The initial target of Cell BE is entertainment – it does not need DP, so Cell BE provides very limited double precision in QS20, QS21, but QS22 support DP with 5 times fast.

IBM use this artitecture for building next generation world fastest super computer Roadrunner Project.

10X faster than normal CPU based cluster 하지만, 고가임.

Cell BE SDK 개발자 거의 없음
- 금융 뿐만 아니라 국내 CS분야에도 극소수
- 게임 개발 분야에 극소수 종사

대안1 Mucury Cell Accelerator

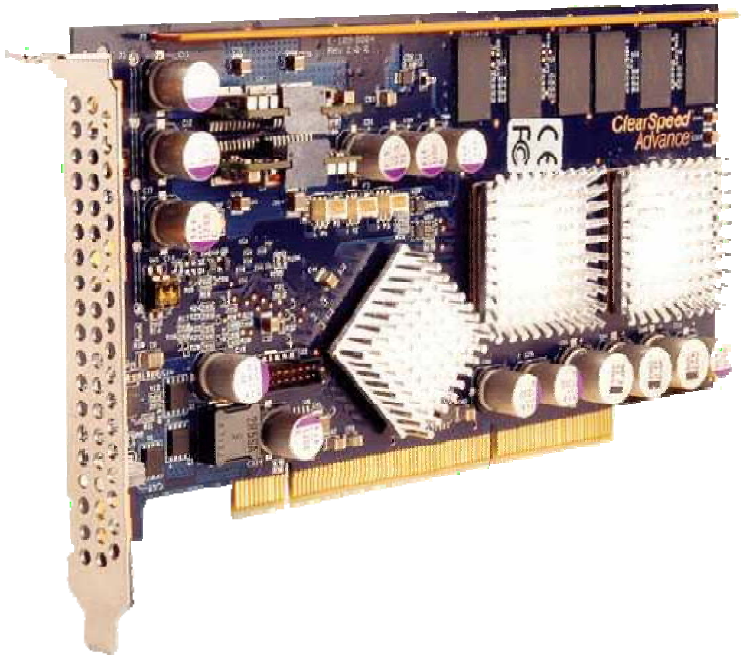
2.8Ghz Cell BE processor in board
One Cell BE has 8 SPE units for computing

Cell BE processor at 2.8 GHz
SP 180 GFLOPS in PCI Express accelerator card
PCI Express x16 interface with raw data rate of 4 GB/s
in each direction
Gigabit Ethernet interface
1-GB XDR DRAM, 2 channels each, 512 MB
4 GB DDR2, 2 channels each, 2 GB
162W for each board

대안2 ClearSpeed CSe620 Accelerate Board

Acceleration Board for HPC
Parallel 192= 2* 96 PE
66 GFlops in Double Precision

Support standard C/C++ SDK
Using CSCN compiler
15W per each board



cheaper than normal cluster solution
4 board system : 300Gflops

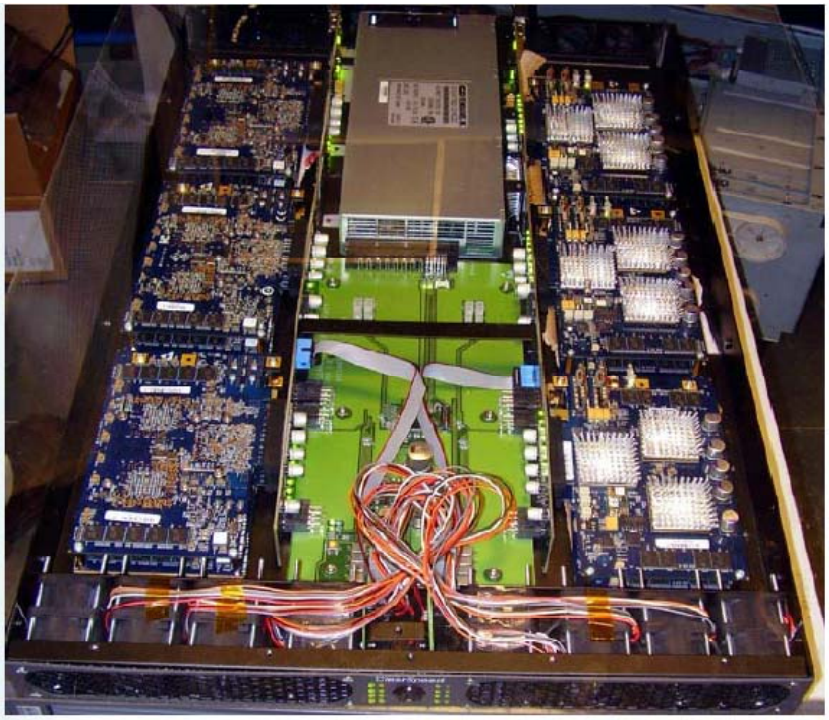
전력소모가 작고, 비교적 발열이 적어 PC, 4U 서버 등에 쉽게 장착 가능

대안2 CATs with ClearSpeed

CS announce CATs in SC07

spec
12 CS board in 1U rack
Power : total 550W

12* 192 Pes = massively parallel
1 DP Tflops



하나의 서버에 최대 12개의 카드 장착가능

대안3 Nvidia Tesla C870, D870, S870

Computing by GPU
Nvidia 8800GTX chipset
Parallel 128 Stream Processors in one chip
500 GFlops in Single Precision

C870 : 1 GPU in board
D870 : 2 GPUs in case
S870 : 4 GPUs in 1U chassis

Today's
3번째Topic

Support standard C/C++ SDK

CUDA

C870 system	SP 500Gflops	: 70 만원
C870 4board system	SP 2Tflops	: 300만원 + PC비용
S870 4GPUs system	SP 2Tflops	: 600 만원+ 서버비용

Cheap & powerful solution

6. 대안적 방법론 I

Clearspeed 가속보드를 이용한 병렬 Monte Carlo Simulation

Co-work with ㈜도우컴퓨팅

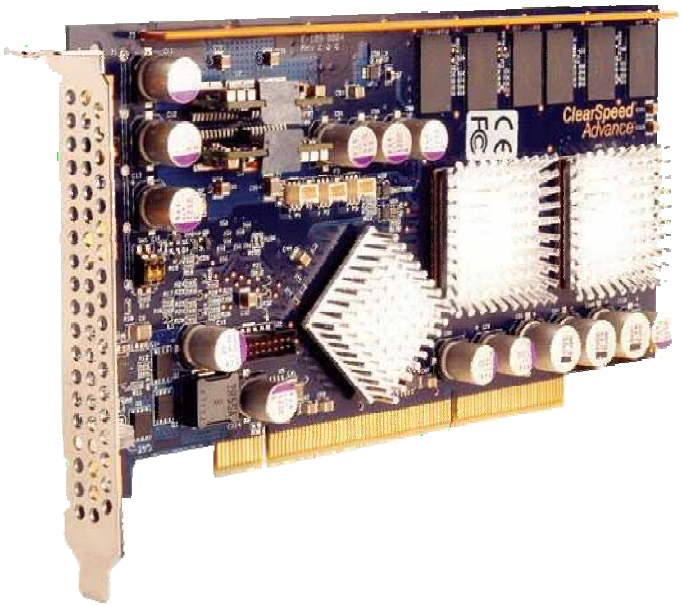
연세대학교
YONSEI UNIVERSITY
금융퀀트연구센터

사용환경

연세금융퀀트연구센터
현재 Clearspeed CSe620 보드 1대를 통해 병렬 시뮬레이션 테스트 중
환경 : Windows XP + Visual Studio 2005+ CSCN 컴파일러 + CSCN SDK

C언어의 확장으로 mono, poly라는 병렬 명령어 셋을 통한 병렬화를 해줘야함

Reference site : 옥스포드 대학에서 LIBOR 쪽 연구



상품종류	2Star Step Down(원금비보장형)		
기초자산	삼성전자(005930) 보통주, SK텔레콤(017670) 보통주		
청약기간	2008년 01월 24일 ~ 01월 24일 오후 1시까지		
청약단위	100만원 이상 10만원 단위	판매한도	10억원
수익발생요건	매 6개월 기초자산의 평가가격이 기준가격의 90/90/85/85/80/80% 이상인(종가기준) 경우		
수익발생기회	만기상환 포함 총6회		
최초기준가격결정일	2008년 01월 24일	만기평가일	2011년 01월 18일
조기상환평가일	최초기준가격결정일 이후 매 6개월		
발행일	2008년 01월 24일	만기일	2011년 01월 24일(3년 만기)
조기상환수익률	8.1% X n차 (연 16.2%)	Knock In 배리어	최초기준가격 X 60%
최대가능수익률	48.6%	최대가능손실률	-100%

Simulation method	Monte Carlo simulation
Method Generating path	G B M (Geometric Brownian Motion) normal distribution
Method of generating normal distribution	Box-Muller method
#of generating path	50000 이상 (기준 100224 개로 작성)
# of Trial (simulation Number)	daily trial
# of SIMD board	Using 1~2 boards

Tao Computing

Development Aim & status

운영 현황
1. ELS (3년 조기 상환형 6 month)
2. 2 Stock Step Down (redeemable Type)
3. knock in barrier 기초자산 60%
4. pricing engine : MC, FDM (SOR type)
5. 개별 평가 Running time 50 sec 이상
6. 운영개발상품 50개
7. 사용자 운영환경 – Windows, C++, Excel Base

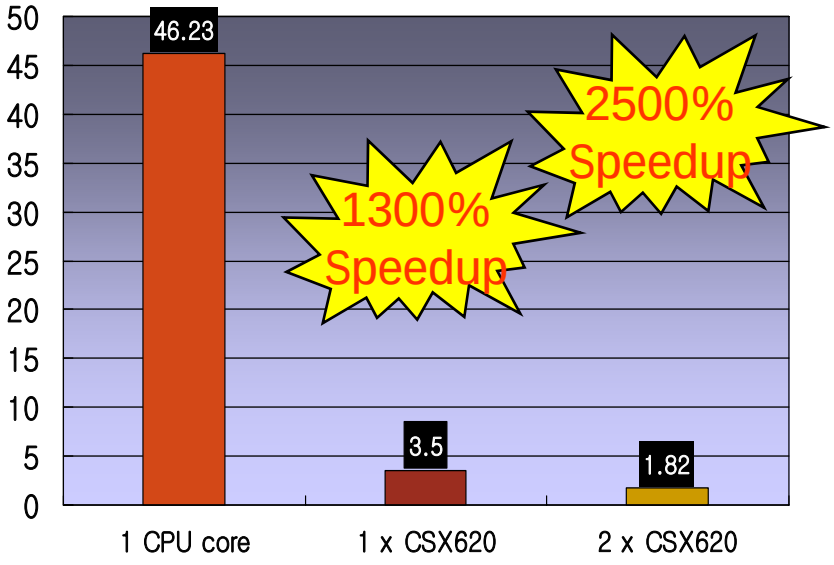
목표
1. 개별 pricing 시간 단축
2. 운영 개별 상품 50개 의 pricing time 단축
3. 사용자 환경유지 : develop platform: c++
   이용환경 :excel base
4. DBMS를 통한 data 중앙 관리
5. ELS 2 stock step down ➔ 가속화
6. Hi five 형 가속화
7. Single stock ELS 개발
8. 가속 Pricing 모델 개발 (ASIAN option etc)

➔ 적정 pricing engine 선정
(5배 이상 ~10 배 속도증가) 6 sec ~ 3sec 이내

target instrument	simulation method	Cal. time	Performance
2 stock step down	MC	46.23	1
(daily observe, # of path 10224)		17.813	2.59
(out put :price, 4 Greek value)		4.562	10.22
gamma 1 ,2		6.906	13.2
delta 1,2		3.21	14.4
		1.82	25.4

Tao Computing

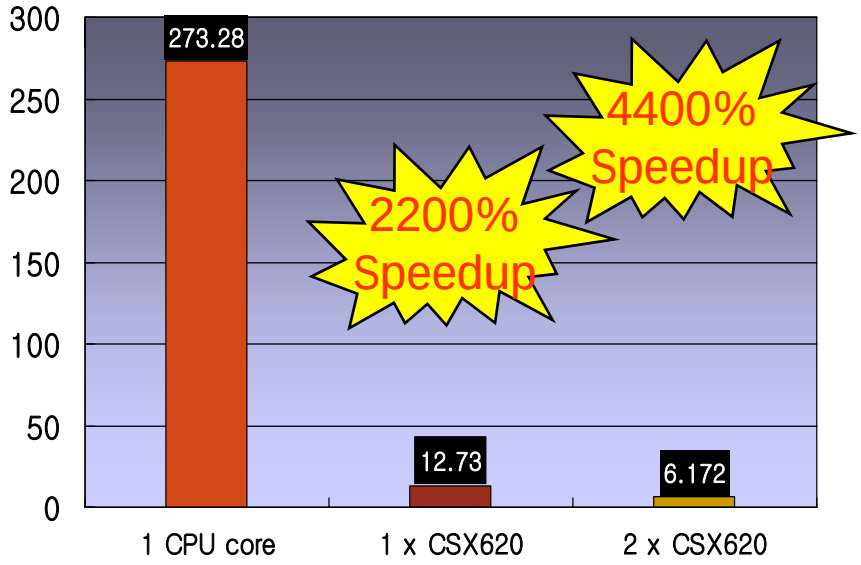
1. 2 Stock Step Down



	1 CPU core	1 x CSX620	2 x CSX620
결과	46.23초 소요	3.5초 소요	1.82초 소요

13배 25배

2. 2 Stock Hi-Five

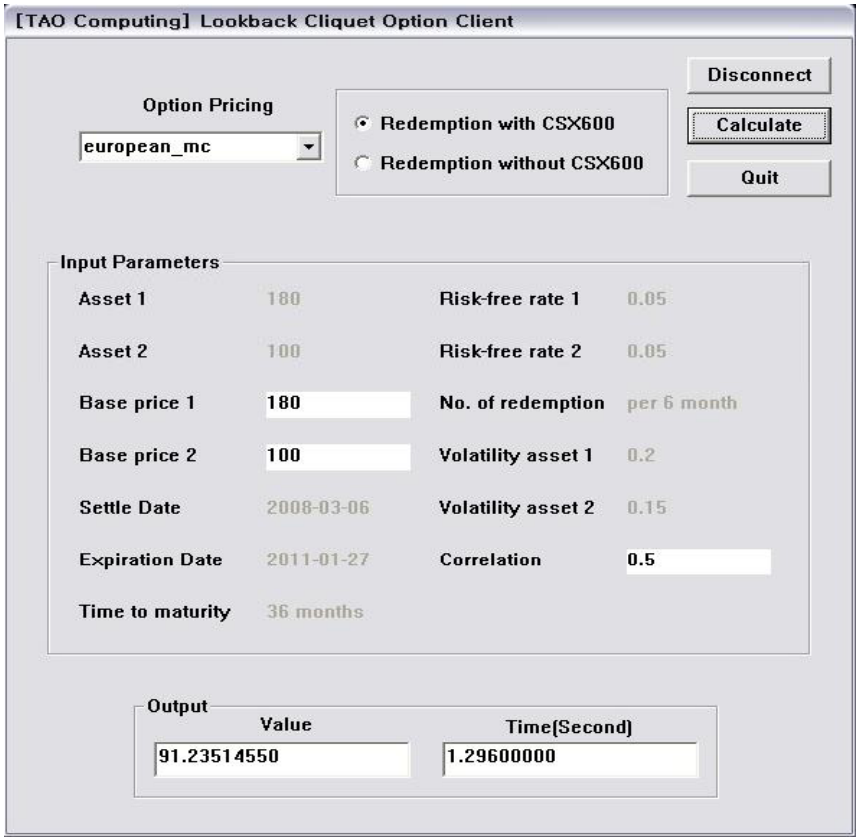


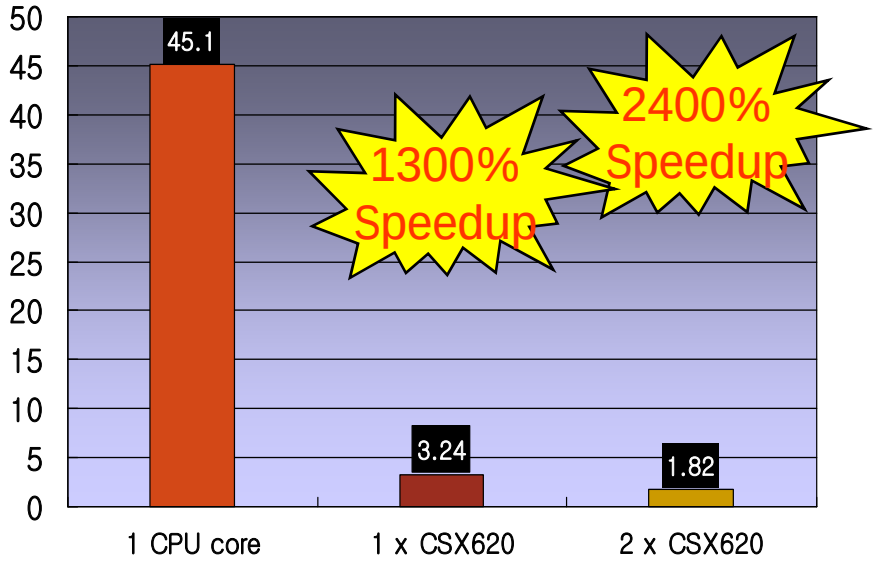
	1 CPU core	1 x CSX620	2 x CSX620
결과	273.28초 소요	12.73초 소요	6.172초 소요

22배 44배

- **3.0 GHz Intel® Xeon® 5130(Woodcrest) Dual core System Base**

• Lookback Cliquet Option





	1 CPU core	1 x CSX620	2 x CSX620
결과	45.1초 소요	3.24초 소요	1.82초 소요

• **3.0 GHz Intel® Xeon® 5130(Woodcrest) Dual core System Base**

Tao Computing

target instrument	simulation method	calculation time		performance
Asian option (path # 152600)	MC	7.115048	serial	1
		0.153944	1 board	47.4
		0.110532	2 board	64.4 배
Asian option (path # 384000)	MC	16.966	serial	1
		0.326242	1 board	58
		0.196844	2 board	85 배
European option (path # 1536000)	MC	5.065038	serial	1
		0.135237	1 board	37.4
		0.098936	2 board	51.68 배
look back cliquet option monthly observe, # of path 1920000	MC	45.1	serial	1
		3.24	1 board	13
		1.82	2 board	24 배
2 stock step down (daily observe, # of path 100224) (out put :price 4 Greek value)	MC	46.23	serial	1
		3.5	1 board	13.2
		1.82	2 board	25.4 배
2 stock Hive Five (daily observe, # of path 96000) (out put :price 4 Greek value)	MC	273.28	serial	1
		12.73	1 board	21.46
		6.1720	2 board	44.27 배

- **3.0 GHz Intel® Xeon® 5130(Woodcrest) Dual core System Base**

7 대안적 방법론 II
CUDA를 이용한
병렬 몬테카를로 시뮬레이션

CUDAp workstation at Yonsei Math-Finance Lab

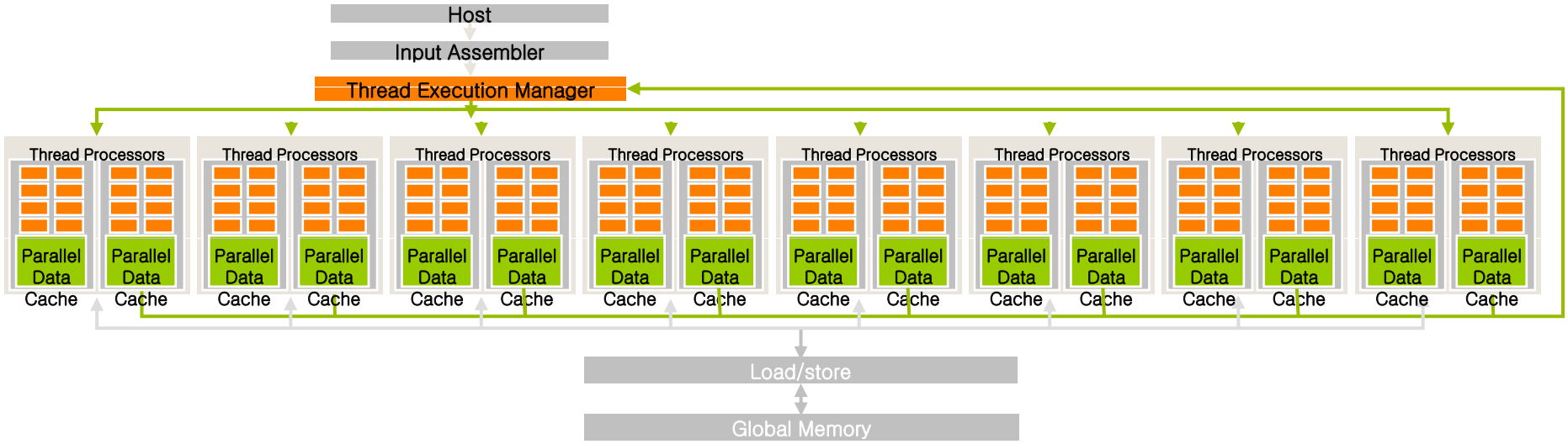
계산용 서버 : Fedora Linux + Intel Compiler + CUDA
Tesla C870 3개 장착 : 이론상 SP 1.5Tflops in SP : 500만원 이하
Tesla D870 1대 추가 도입예정

보조용 워크스테이션 Windows XP + Visual Studio 2003 + CUDA
Tesla C870 1개 장착 : 이론상 SP 0.5Tflops in SP

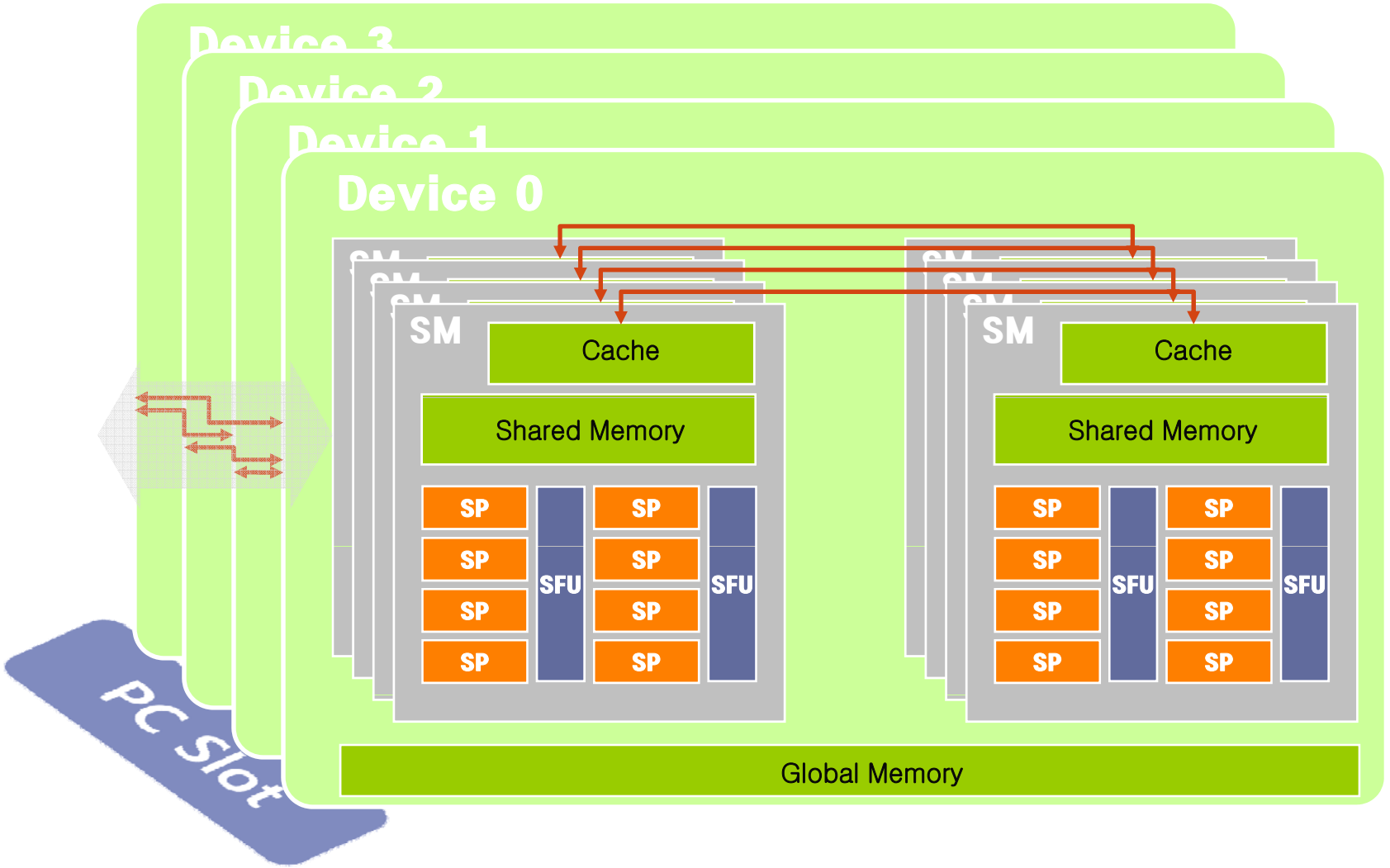
비교
Kisti 수퍼컴퓨터 4호기 : 전체 : DP 24Tflops : (40억원 이상),
하나의 노드 : DP 0.1Tflops : 2천만원 이상

G80 Device in Tesla Inner

- Processors – execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors grouped into 16 Multiprocessors (SMs)
- Parallel Data Cache (Shared Memory) enables thread cooperation

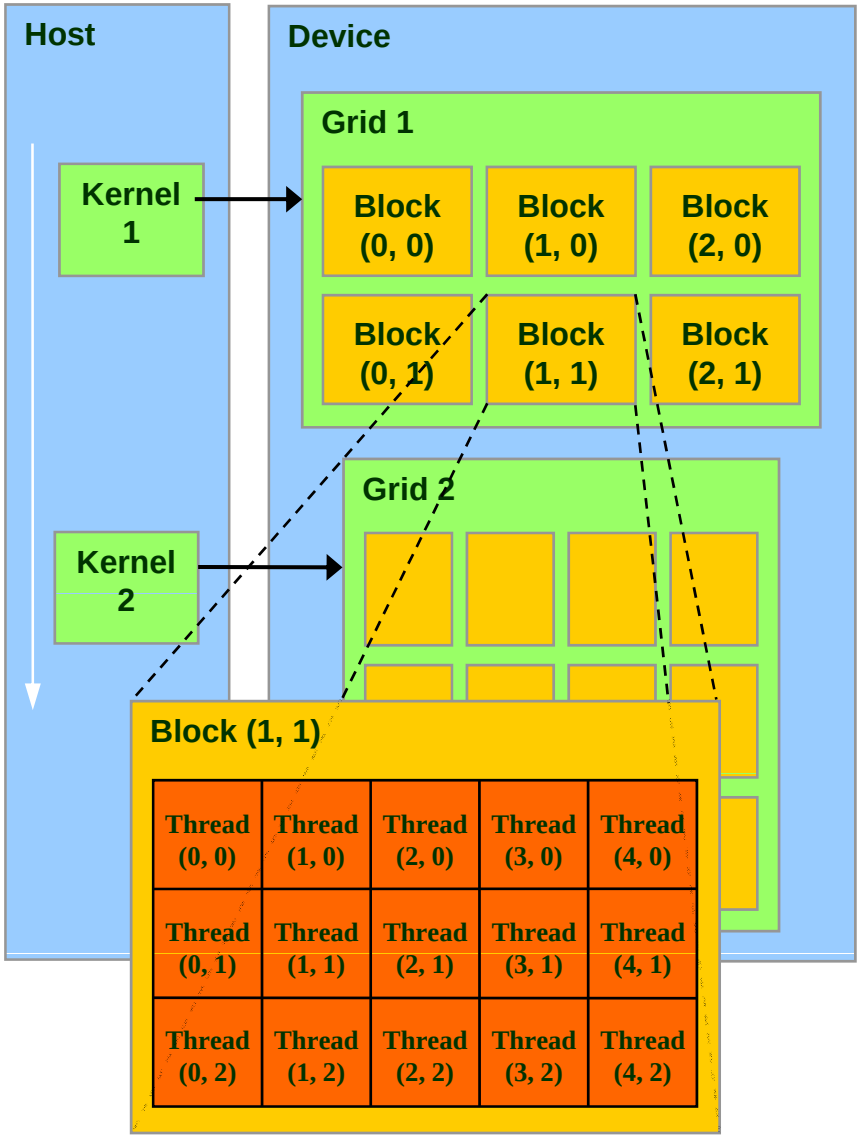


G80 Device In Tesla H/W

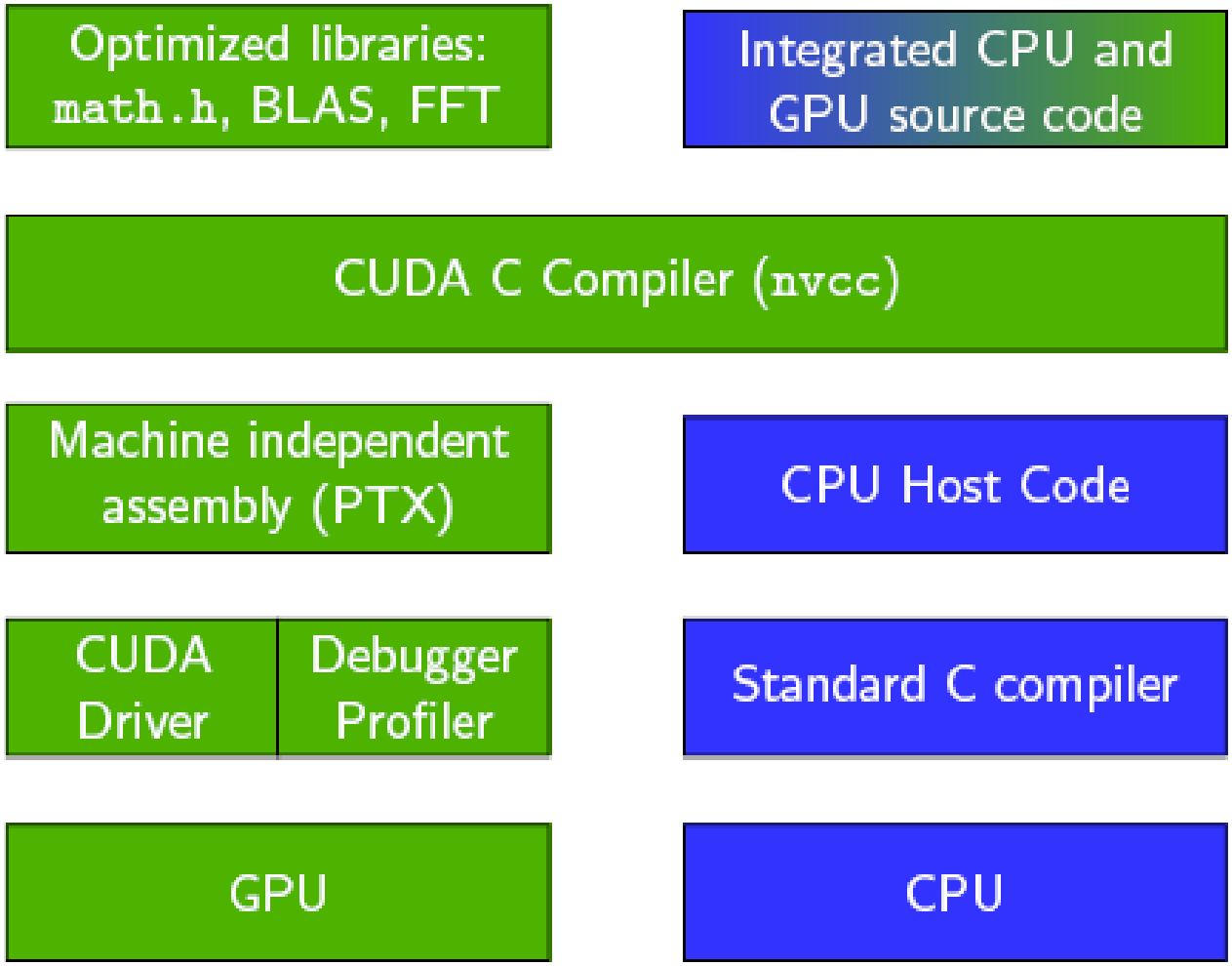


SDK architecture

Threads blocks for SMs
SM launch Waps of threads



CUDA compiler



CUDA syntax

Function_name <<<a, b, c>>>(variables)

CuMalloc(A,size)

CuMemcpy(A,B,method)

__global__ function (variable) { }

__device__ function (variable) { }

__share__ variable;

__ global__ variable;

CUDA algorithm example

single

```
void add_matrix
( float* a, float* b, float* c, int N ) {
    int index;
    for ( int i = 0; i < N; ++i )
        for ( int j = 0; j < N; ++j ) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
    }

int main() {
    add_matrix( a, b, c, N );
}
```

CUDA

```
__global__ add_matrix
                ( float* a, float* b, float* c, int N ) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        int index = i + j*N;

        if ( i < N && j < N )
        c[index] = a[index] + b[index];
       }

int main() {
    dim3 dimBlock( blocksize, blocksize );
    dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

Single Precision vs. Double Precision

Single Precision : 32bit
Error : 24 mantisa
10^-7

double Precision : 64bit
Error : 24 mantisa
10^-15

Round-off errors in single precision (example in Knuth)
(11111113. **[+]** - 11111111.) **[+]** 7.5111111 = 2.0000000 **[+]** 7.5111111 = 9.5111111.
11111113. **[+]** (- 11111111. **[+]** 7.5111111) = 11111113. **[+]** - 111111103. = 10.000000.

Financial Monte Carlo simulation use math function of sqrt, ln, exp, sin, cos or numerical approximation for transformation and cdf
Random number generator use divide

In this situation , with 10^-7 rounding
we will generate same r.v. in different values and lose the accuracy

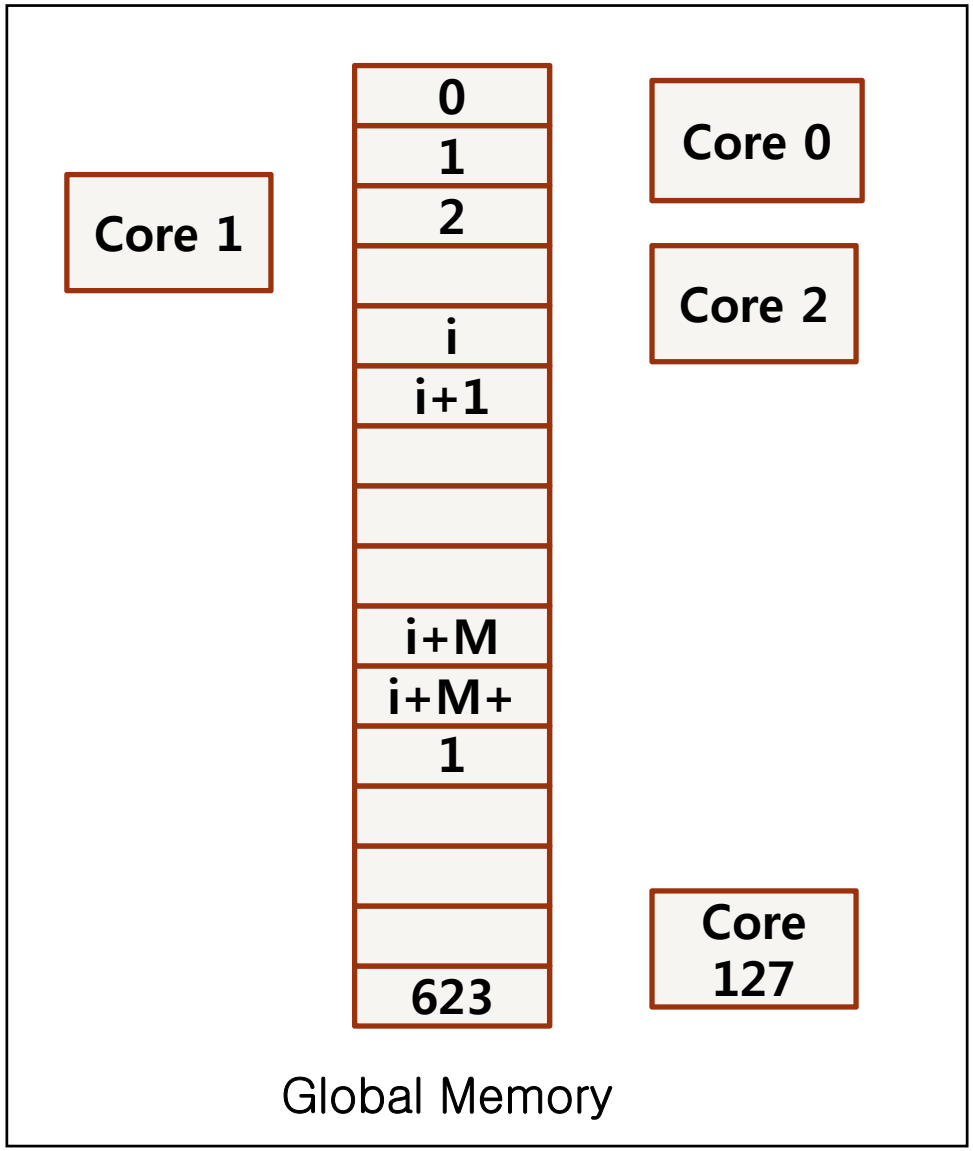
But, in Financial Industry,
they allow 1BP in Hedge Vol : that means they allow 10^-4 errors in price.
If we can control the errors 10^-5 in single precision in MC. (not FDM, FEM)

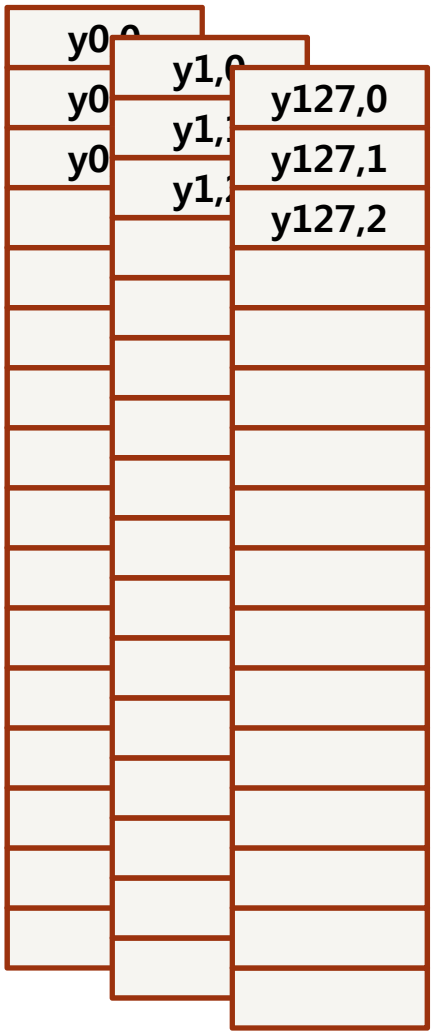
Implementation for pMC on CUDA

Massively pMC (H/W)

- 다양한 문제를 고민해줘야함
- RNG 자체의 분석 필요 (SMT19937 분석)
- 비 금융적 문제임
- Parallel H/W에 관련된 문제가 대부분
- Knuth, the art of computer programming

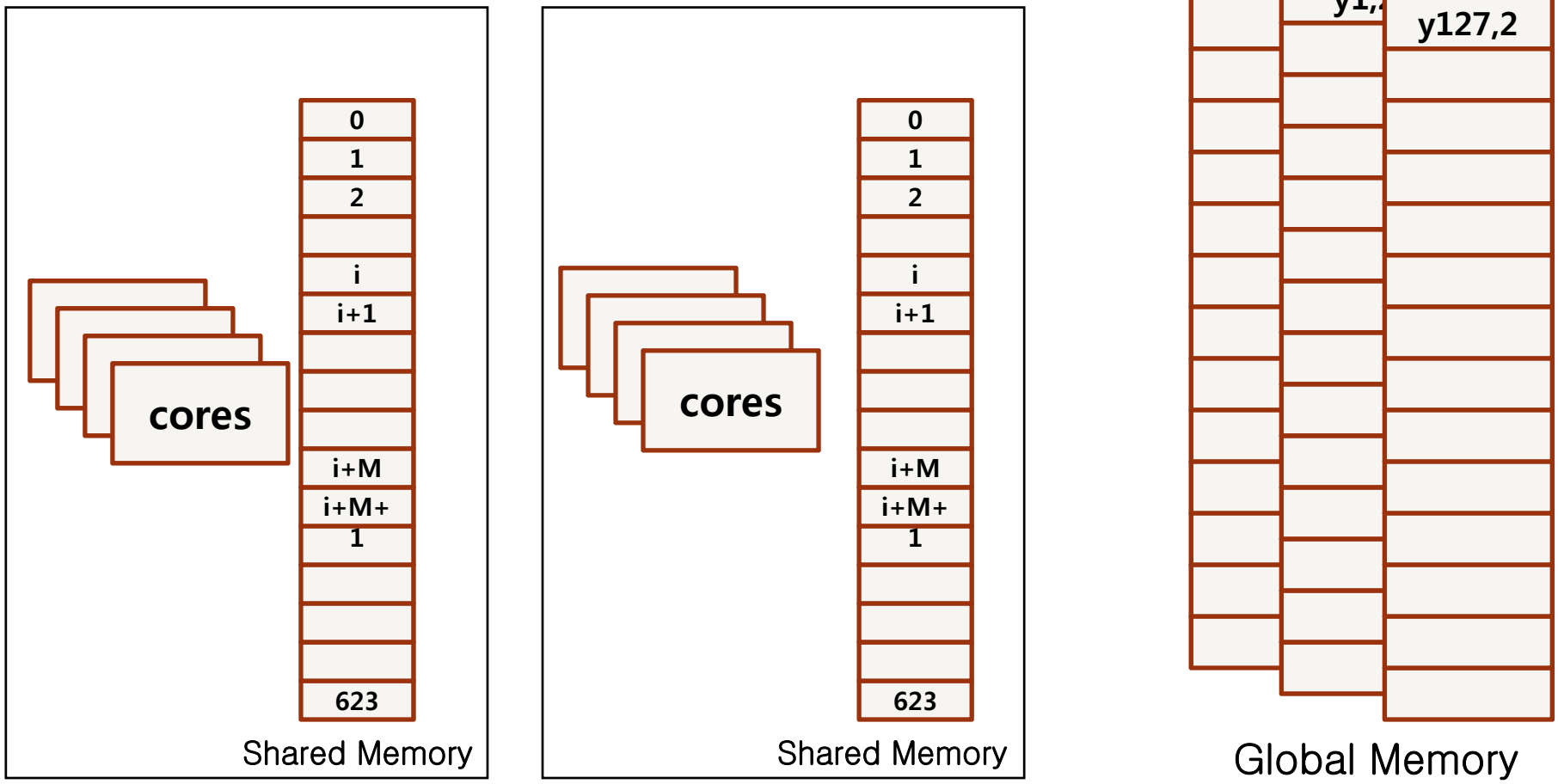
CUDA 방법I -- PMT19937





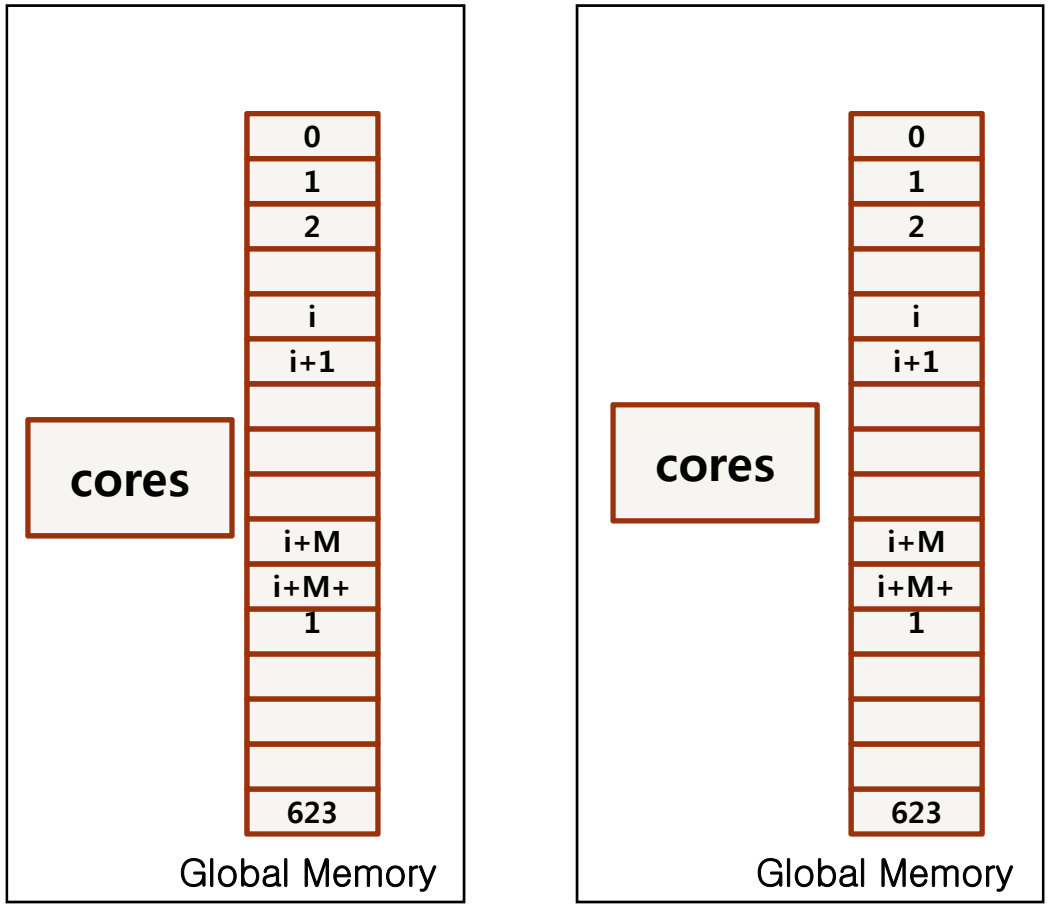
CUDA방법 II -- PMT19937

각 16 Block은 Multi Seed 이용



CUDA 방법III -- parallel SMT

각 128 SPs가 Multi Seed 이용



New approach for massively parallel SPMD

1. **Job split method**
   Simulate Option Price in Outer Loop.
   each simulation, call RNG
   it has big overheads of function call
   Wall Clock Time >> Simulation Time / (# of cores)

2. **Pre-generation method**
   generate full # of U R.V. for simulation
   Transfer U R.V. to N R.V.
   each simulation, they use R.V.s which were already generated.
   limit of memory bandwidth : 60GB/s : needs 0.025 sec for 1.5GB
   limit of memory size : 2억개의 R.V.s /each device
   Wall Clock Time = Simulation Time / (# of cores) + Memcpy Time

3. **Avoid Round-off Errors**

$$\mathrm{E}[e^{-r\tau}[S_T - K]^+] \approx \frac{1}{M}\sum_{k}^{M}\left(\frac{1}{N_k}\sum_{i}^{N_k} e^{-r\tau}[S_{T_i} - K]^+\right)$$

New approach for massively parallel SPMD

4. **Mixed Precision Method (float-float approach)**
   emulate double precision with two single precision operation
   DP is 4~10 times slower than SP.
   DP RNGs much more slower than SP RNGs
   - same period but : 24bit or 48bit mantisa in SP, 54bit mantisa in DP
   - heavy function call of ln, sin, cos in Box-muller

5. **Mixed Precision Method (GPU-CPU approach)**
   use 64bit CPU FP in DP computation
   It need comuniticating between Host and Device.
   DP in CPU is slower than DP in GPU
   Tt is easy to imply

6. **Fixed-Point Method**
   convert float point data to integer data.
   ALU is faster than FP
   Fixed Point Method is useful in LCG, but is not suitable in Box-muller

New approach for massively parallel SPMD

7. **Inter-correlation between parallel cores**
   split wthout careful considering,
   the inter-correlation between parallel cores occure.
   To avoid this, we need to parallelize with considering algorithms of RNG
   for different RNGs, we apply different method of parallelizations

8. **Overlapping in splitted random sequence**
   Pseudo RNGs has limited period such as 2^31, etc.
   In massively parallel generating,
   each stream of random number may overlapp with each others.
   To avoid this, use long period RNGs or dynamic creation method.

Benchmark for RNGs with CUDA

Rand48() + Memcpy(DtoH)

CPU : 최신 AMD 페놈 2.5Ghz
GPU : Tesla C870

Size: 122880000 random numbers

CPU rand48

time : 2260.000000 ms

Samples per second: 5.2512821E+07

GPU rand48

time : 20.000000 ms

Samples per second: 6.144000E+09

Copying random GPU data to CPU

time : 540.000000 ms

속도향상 : 최대 113 X

Thread에 따라 80X 정도

Benchmark for RNGs with CUDA

MT19937 + Box-Muller + Memcpy

Size: 122880000 random numbers

CPU MT19937 RN generation

time: 1460.000000 ms

Samples per second: 8.4164384E+07

Size: 122880000 random numbers

GPU MT19937 RN generation

time : 199.955994 ms

Samples per second: 6.145352E+08

속도향상 : 7.3 X

원래 MT알고리즘이 LCG보다 속도 빠름
LCG 2260ms vs. MT 1460ms

그런데 가속화 성능은 LCG가 더좋음
LCG 113X vs. MT 24X
LCG 20ms vs. 48ms

Generated samples : 100007936

RandomGPU() time : 47.960999 ms

Samples per second: 2.085193E+09

속도향상 : 24.7 X

Occupacy for thread & Performence

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	760
Active Warps per Multiprocessor	24
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	100%
Maximum Simultaneous Blocks per GPU	64

(Note: This assumes there are at least this many blocks)

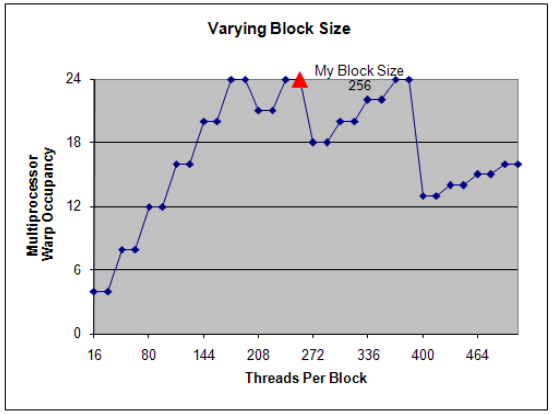
Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384

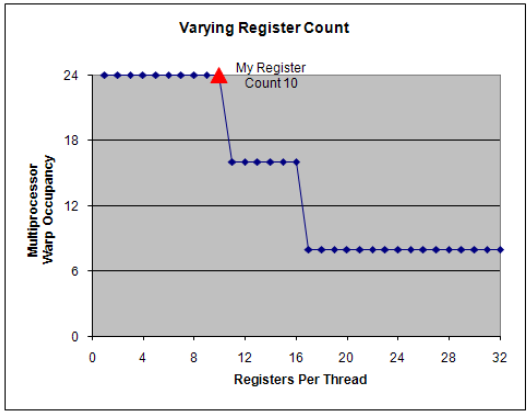
Allocation Per Thread Block	
Warps	6
Registers	1920
Shared Memory	4096

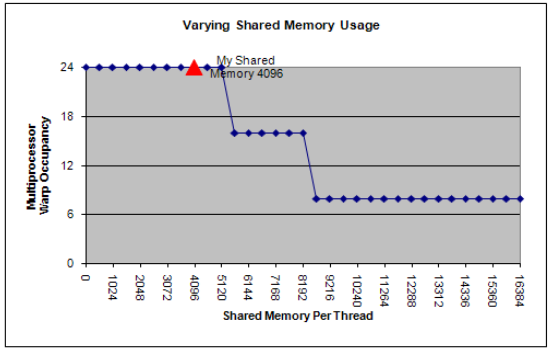
These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	4
Limited by Shared Memory / Multiprocessor	4

Thread Block Limit Per Multiprocessor is the minimum of these 3







<<<16,256>>>

Performance from Operation Cycles & latency

Operation
type conversion : 4 cycle

float add/mul/mad : 4 cycle
floating div : 36 cycle

integer add, bit operation : 4 cycle
integer comare, min, max : 4 cycle
integer mul : 16 cycle

Memory
4 cycles for issuing a read from global memory
4 cycles for issuing a write to shared memory

400~600 cycle for reading a float from global memory
→Need thread schedule strategy

Benchmark for European Vanilla

GPU MT19937 RN generation

time : 199.955994 ms

Samples per second: 6.145352E+08

GPU BoxMuller transformation

time : 98.652000 ms

Samples per second : 1.245591E+09

Copying random GPU data to CPU

time : 517.033020 ms

GPU Monte-Carlo simulation...

Total GPU time : 56.948002 ms

Options per second : 1.755988E+01

Total : 872.589 ms = 0.8 sec Non-full path generation

Implementation on ELS pricing and hedging

상품설명서 : **삼성증권 제1909호 주식연계증권**

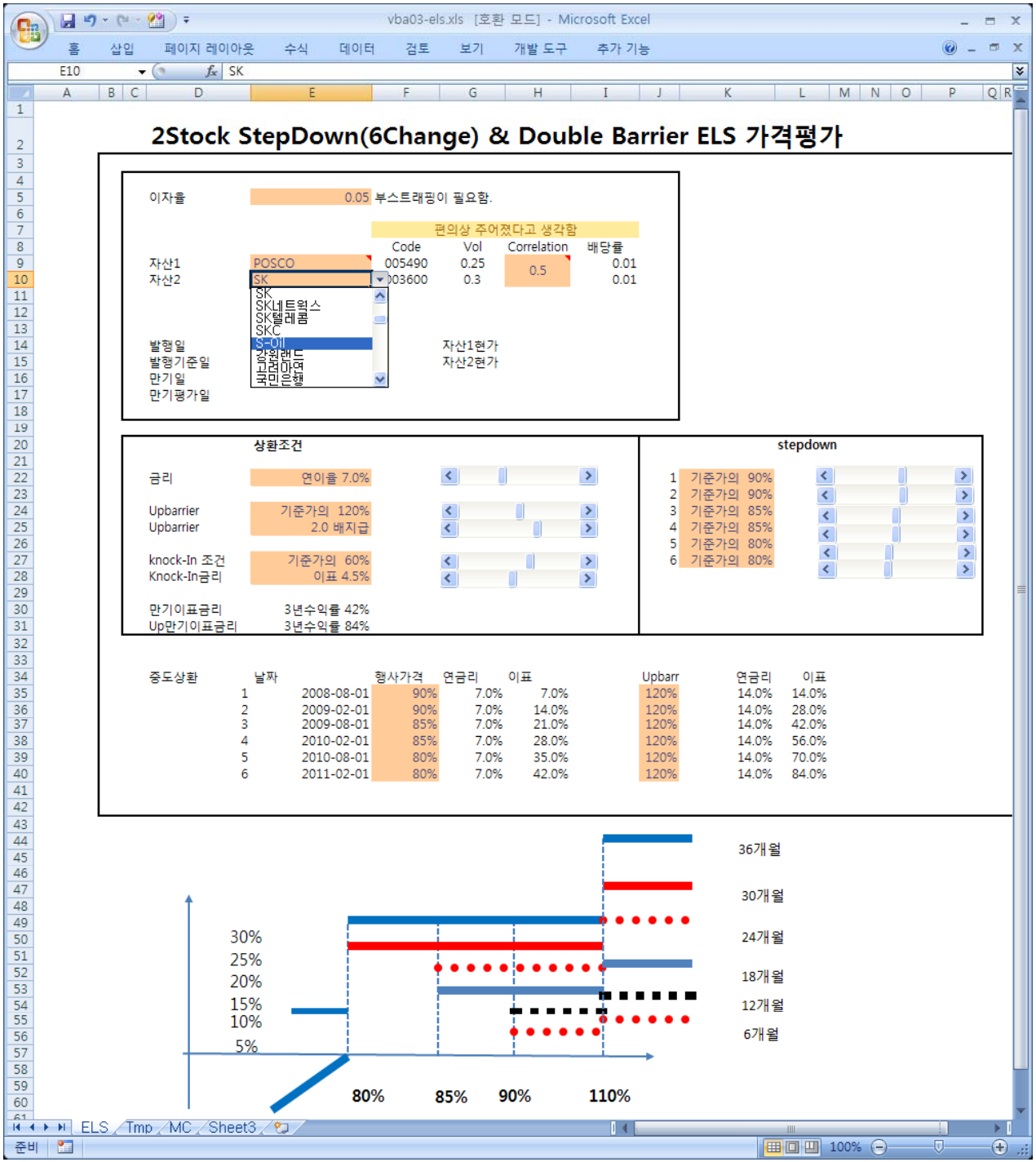
3년 만기
2 star : POSCO 일반주, S-Oil

12 chance : 디지털 옵션

Double Barrier
Down barrier : 장중체크 (둘중 하나라도 하락한계)
Up barrier : 매일 종가 체크(두개다 상승한계)

Step-down : 없음

지급 : 조기상환시 +2 영업일



Excel 화면

삼성증권 1909회 ELS (95형)

face value **10000**

	POSCO	S-Oil	비율
최초기준가	539000	68700	
행사가격	512050	65265	95%
상승한계	549780	70074	102%
하락한계	323400	41220	60%
현재가	539000	68700	
발행일	2008-05-13	기간	
만기	2011-05-09	3년	
현재날짜	**2008-05-14**		

Volatility 추정

	POSCO	S-Oil
Hvol	0.410461777	0.373543706
EWMA	0.385880533	0.448304863

Correaltion 추정

0.328060489

할인율 0.06

조기상환일		days	networkdays		지급 : 년12%	처음부터	처음부터	현재부터
1차	2008-08-08	64	61		0.03	60	366	360
2차	2008-11-07	129	123		0.06	122	738	732
3차	2009-02-09	195	185		0.09	184	1110	1104
4차	2009-05-08	259	247	1년	0.12	246	1482	1476
5차	2009-08-07	324	310		0.15	309	1860	1854
6차	2009-11-09	390	375		0.18	374	2250	2244
7차	2010-02-09	456	439		0.21	438	2634	2628
8차	2010-05-07	519	498	2년	0.24	497	2988	2982
9차	2010-08-09	585	563		0.27	562	3378	3372
10차	2010-11-09	651	626		0.30	625	3756	3750
11차	2011-02-09	717	689		0.33	688	4134	4128
만기	2011-05-09	780	749	3년		748	4494	4488

Excel/VBA Pseudo code for ELS (single)

Parameter 입력

Loop 10만번~ 50만번
    XT simulation 실행
        Boxmuller,
        RNG(SMT19937)
    ELS pricing Routine
        IF문을 통해 조기사환, 만기조건 고려
        조기상환시 시뮬레이션 정지

평균값을 구함

결과 출력

CPU single 실행속도

- 엑 셀 : 50만번 시뮬레이션(Pricing만) : 약 4~5분(조기 상환되었는데도 느림)

  → Excel이 실행되는 CPU는 성능이 안좋음.

시스템 : Intel Core2Duo 2.0Ghz dual core (but single using)

(장중 모니터링 6회 실시) 즉 50만번 시뮬레이션 시

총 사용 RNG 개수 :500000*250*3*6*2 = 4500000000 개 : 45*10^8

조기상환이 되도 계속 RNG를 생성시킨 경우(비교를 위해)

C MT19937 이용시		최적화		최적화중
RNG	CPU 53초,	GPU1 7.3초,	GPU2 2.1초	
BM 변환	CPU 20초,	GPU1 3.6초,	GPU2 1.1초	
Copy	CPU 0초 ,	GPU1 18.3초,	GPU2 6.2초	
MC	CPU 18초,	GPU1 1.2초,	GPU2 0.4초	
총	90초,	30초(2.5배)	9.8초(9배)	7.1초(12배)

C언어 : AMD 페놈 2.5Ghz, GPU : Tesla C870

CPU single 실행속도

대부분 1-2 회차에서 조기상환됨

총 사용 RNG 개수 : 500000*120*6*2 = 약 720000000개

전체 계산량의 1/6로 감소 : 이론상 CPU에서 15 sec 가능??

불가능함.

원인 : Function call에 의한 오버헤드 (IF문 사용)로 인한 속도 증가 어려움

병렬처리시에도 비슷한 문제 발생, 더욱 복잡한 문제들 발생가능성 있음.

VBA -> C/C++ 변환

Excel의 networkdays 함수를 직접 구현해주어야함.
(어려운 점 : 실무 경험이 부족하여 1일 이상 소요 아직도 완벽하지 않음)

배열을 사용시 언어의 특징 고려해야함
variable(i) → variable[i]
Option base 1을 option base 0 로 바꿔줘야함

For문, if문의 형식을 맞춰줘야함. 특히 bracket { } 에서 오류 많이 발생
특히 연산자 &&, ||, !, == 오류 주의

대소문자 오류 체크 (C언어는 변수명과 함수명의 대소문자에 민감함)

C언어 DLL을 이용한 excel link도 사용가능

C언어 Single -> CUDA 병렬 코딩

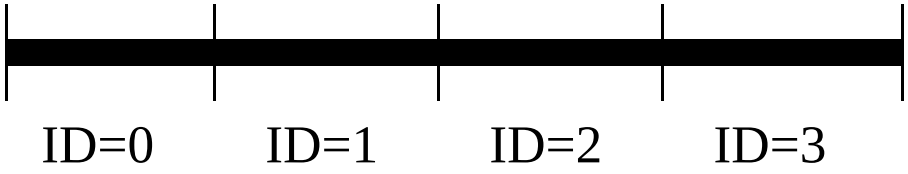
병렬 MC아이디어를 사용

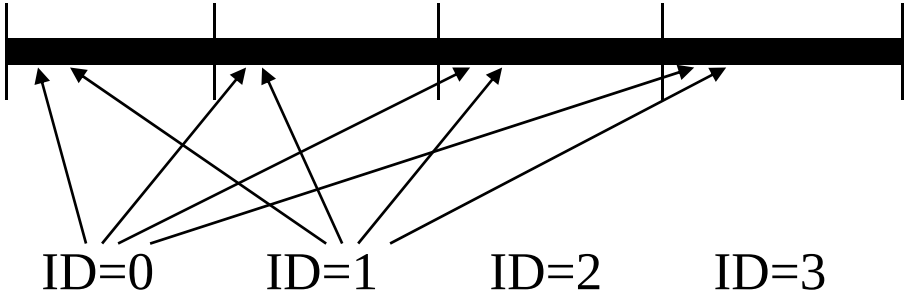
CPU영역과 GPU 영역을 정확히 구분해줘야함

하드웨어 및 CUDA 자체의 특징을 고려해 줘야함.
thread 고민
memory 고민

Thread control on SPMD algorithms

How do we divide job (FOR LOOP) ?





Thread control on SPMD algorithms

1

```
__Global__ ELS (parameters){
   Tid= blockIdx.x*blockDim.x + threadIdx.x;
   N=blockDim.x *threadDim;
    for (i = 0; i < Nsim/N; i++) {
       ….
   }
}
```

2

```
__Global__ ELS (parameters){
   Tid= blockIdx.x*blockDim.x + threadIdx.x;
   N=blockDim.x *threadDim;
   int rem = Nsim %   N;
   i_start = Tid  * (Nsim /N);
   i_end = i_start + (Nsim /N);
   if (Tid == (N-1))  i_end = N;
   for (i = i_start; i < i_end; i++) {
       ….
   }
}
```

3

```
__Global__ ELS (parameters){
 Tid= blockIdx.x*blockDim.x + threadIdx.x;
   N=blockDim.x *threadDim;
   for (i = Tid; i < Nsim; i+= N) {
        ….
   }
}
```

1,3 method is good

3 method is recommanded

CUDA 고급 메모리 관리

CUDA의 제한사항

__global__, __device__ 함수 내부에서는 static 변수의 선언 불가

Pointer는 global 메모리영역만 지정 가능

global 함수는 무조건 void 함수임.

사용방법

__global__ 함수 밖에서 static 변수를 미리 정의해 줘야함.

__global__ 에서는 메모리를 참조해야함.

Global memory의 포인터 주소를 참조

사용 예)

```
Main(){
__device__  static mt[624*N];                          global memory에 할당됨
__device__  __shared__ static mti,bmused,y1,y2,r1,r2;
__device__  static seed;

MT19937 <<<32,8>>>(void)
}
__kernel__  MT19937(){
Mt[block*BlockID+ 0]=seed;
…
}
```

CUDA 고급 메모리관리 shared pMT19937구현시

Nvidia G80 chip 스펙 16 KB
on chip shared memory : 16384 bytes per Block (1 cycle)
on device global memory : 512~1.5 GB (200 cycle)

Shared Memory가 상당한 고가임

필요한 Static Variables
1개의 SMT19937 처리시 필요한 변수들
배열 : MT[624] - 19968 bytes
변수 : mti, bmused, y1,y2,r1,r2 – 192 bytes

20 KB

배열만으로 H/W제공 shared memory size보다 많이 필요

→ Shared memory(fast)가 아닌 Global memory(slow)사용
mti,bmused, y1,y2,r1,r2 등은 shared memory 사용가능

→ Global memory 사용에 의한 Bottleneck 발생 5배 이상 속도저하

CUDA 차세대 버전에서 필요한 Memory Size

차세대 Nvidia H/W에서의 필요 shared memory size

1차 21504 byte per Block 지원시
고속 shared pMT19937 구현가능

```
__shared__ static mt[624];
__shared__ static mti,bmused,y1,y2,r1,r2;
__device__ static seed;
```

차세대 GPU에서 Global Memory를 DDR5로 제공하여 bandwidth를 높이는 경우에도 추가적인 속도 향상 기대됨

2차 161280 byte per Block 지원시
고속 fully pMT19937 구현 가능

```
__device__ static mt[624*N];
__shared__ static mti,bmused,y1,y2,r1,r2;
__device__ static seed;
```

Case I Algorithm for ELS

Parameter 입력

Loop 병렬화
    RNG(PMT19937) -- shared / global memory version
    Box Muller 실행

Loop 병렬화
    XT simulation 실행 <- 만들어진 RNG 사용함
    ELS pricing Routine
        IF문을 통해 조기사환, 만기조건 고려

평균값을 구함

결과 출력

Case I　CUDA구조

템플릿

```
Main(){
Parameter & memory copy;
ELS_bodyGPU<<<A,B>>>();
get results;
Print results;
}

__global__ ELS_bodyGPU(){
ELS_kernel();
}

__device__ ELS_singleGPU(){
Option pricing algorithm;
}
```

모듈화

```
__device__  boxmuller(){
}
__device__  MT19937(){
}

void ymalloc() {
}
void yhtod() {
}
void ydtoh() {
}
void ytstart() {
}
```

Case I Pseuco code for CUDA ELS

```
Main(){
   cudaMalloc((void**)  &MTd, 624*sizeof(float));
   cudaMalloc((void**)  &a, 1024*1024*2*sizeof(float));
   Dim3 DimGrid();
   Dim3 DimBlock();
   Random_GPU <<<DimGrid,DimBlock>>>  (parameters);
   Boxmuller_GPU <<<DimGrid,DimBlock>>>  (parameters);
   ELS <<<DimGrid,DimBlock>>>  (parameters);
   sum( option[k] ) /N;  //N is 128

}
__Global__ ELS (parameters){
   Tid= blockIdx.x*blockDim.x + threadIdx.x;
   N=blockDim.x *threadDim;
For( I<0, I< Nsim/N;I++){
      For(j<0,j<Totalday){
         For(k=0;k<monitor;k++){
            Norm1=a[N*i+ blockIdx.x*blockDim.x + threadIdx.x];
            Norm2= a[N*(i+N)+ blockIdx.x*blockDim.x + threadIdx.x];
            Xt1(I)= Xt1+MuT*dt + SigmaT*Norm1
            Xt2(I)= Xt1+MuT*dt + SigmaT*Norm2
            if (Xt1(I) <DB1 and Xt2(I) <=DB2 ) down_flag=1;
            }
         if (Xt1(I) >DB1 and Xt2(I) >=DB2 ) up_flag=1;
         if(j=pre1){}
         if(j=pre2){}
         }
         option = ;
         sum = sum+option;
         option = 1/(Nsim/N)*sum;
   Return option[tid];
}
```

```
__device__  float Box_Muller( float a1, float a2){
r-=sqrt(-2.0f *logf(u1));
Float phi = 2*PI*u2;
a1=r*__cosf(phi);
a2=r*__sinf(phi);
}

__ device__ float BoxMuller_GPU(){
Return Box_Muller( a[N*i+Tid], a[N*(i+1)+Tid );
}


__ device__ random_GPU(){
//Use static variables for each threads
Algorithms for MT RNG

a[K*i+Tid]=y;  //K=2N
}
```

Case I 분석

Random Number Generation이 대부분 ELS pricing의 계산시간을 차지

장점
CUDA를 이용한 가장 빠른 parallel RNG 생성기법임
함수call에 의한 오버헤드 발생이 거의 없음

단점
미리 RNG를 생성하기 때문에 Memory Size의 제약을 받음
RNG를 Global Memory에 생성후 Xt생성시 Global Memory에서
불러와야 하기 때문에 계속적인 Memory bottle neck 발생

Critical한 단점
조기상환의 경우 미리 만들어 놓은 RNG를 사용할 필요없음
조기상환형 문제 해결을 위한 방법을 고민해야함

→ 조기상환형의 경우 속도향상을 기대하기 어려움 (현재)
조기상환형 상품을 Case I의 알고리즘으로 구축시 GPU를 사용할 필요가 없음

조기상환일별 RNG를 재추출 하는 방법으로 속도향상기대 (코딩이 복잡해짐)

Case II Algorithm for ELS

Parameter 입력

Loop 병렬화
    XT simulation 실행
        Boxmuller,
        RNG(parallel SMT19937)
    ELS pricing Routine
        IF문을 통해 조기사환, 만기조건 고려

평균값을 구함

결과 출력

Case II Pseuco code for CUDA ELS

```
Main(){
   cudaMalloc((void**)  &MTd, 624*sizeof(float));
   Dim3 DimGrid();
   Dim3 DimBlock();
   ELS_body <<<DimGrid,DimBlock>>>  (parameters);
   sum( option[k] ) /N;  //N is 128
}

__global__ ELS_ body(parameters){
 Tid= blockIdx.x*blockDim.x + threadIdx.x;
  N=blockDim.x *GridDim;
Initialize();
ELS_ kernel(parameters);
}
```

```
__ device__   Initialize(){
Initialize DC of MT19937
}
__ device__   float Box_Muller(){
U1= MyRand();U2= MyRand();
If( used =1){ return} else {return }
}
```

```
__device__ ELS_ kernel(parameters){
For( I<0, I< Nsim/N;I++){
      For(j<0,j<Totalday){
         For(k=0;k<monitor;k++){
            Norm1(i)=Box_Muller();
            Norm2(I)=Box_Muller();
            Xt1(I)= Xt1+MuT*dt + SigmaT*Norm1
            Xt2(I)= Xt1+MuT*dt + SigmaT*Norm2
            if (Xt1(I) <DB1 and Xt2(I) <=DB2 ) down_flag=1;
            }
         if (Xt1(I) >DB1 and Xt2(I) >=DB2 ) up_flag=1;
         if(j=pre1){}
         if(j=pre2){}
         }
         option = ;
         sum = sum+option;
         option = 1/(Nsim/N)*sum;
   Return option[tid];
}
```

```
__ device__   float MyRand(){
Return MT19937()/4294967296.0;
}
__ device__   float MT19937(){
//Use static variables for each threads
Algorithms for MT RNG
Return y;
}
```

SPDM1 방법론 사용

Case II single 코드의 재사용

```
Main(){
   cudaMalloc((void**)  &MTd, 624*sizeof(float));
   Dim3 DimGrid();
   Dim3 DimBlock();
   ELS_body <<<DimGrid,DimBlock>>>  (parameters);
   sum( option[k] ) /N;  //N is 128
}

__global__ ELS_ body(parameters){
 Tid= blockIdx.x*blockDim.x + threadIdx.x;
  N=blockDim.x *GridDim;
Initialize();
ELS_ kernel(parameters);
}
```

```
__ device__   Initialize(){
Initialize DC of MT19937
}
__ device__   float Box_Muller(){
U1= MyRand();U2= MyRand();
If( used =1){ return} else {return }
}
```

Template과 모듈이용시
적색은 부분만 코딩해주면 됨
Single code와 동일한 구조

```
__device__ ELS_ kernel(parameters){
For( I<0, I< Nsim/N;I++){
      For(j<0,j<Totalday){
         For(k=0;k<monitor;k++){
            Norm1(i)=Box_Muller();
            Norm2(I)=Box_Muller();
            Xt1(I)= Xt1+MuT*dt + SigmaT*Norm1
            Xt2(I)= Xt1+MuT*dt + SigmaT*Norm2
            if (Xt1(I) <DB1 and Xt2(I) <=DB2 ) down_flag=1;
            }
         if (Xt1(I) >DB1 and Xt2(I) >=DB2 ) up_flag=1;
         if(j=pre1){}
         if(j=pre2){}
         }
         option = ;
         sum = sum+option;
         option = 1/(Nsim/N)*sum;
   Return option[tid];
}
```

```
__ device__   float MyRand(){
Return MT19937()/4294967296.0;
}
__ device__   float MT19937(){
//Use static variables for each threads
Algorithms for MT RNG
Return y;
}
```

Case II 분석

장점
RNG 저장에 메모리를 사용하지 않아 이론상 최대 주기까지 RNG 생성 가능
조기상환시 더 이상 RNG를 생성 안함
Single Program 알고리즘을 거의 수정하지 않고 사용할 수 있음
→ 모듈화 가능

단점
하나의 코어에서 SMT를 돌리기 위해 필요한 메모리 용량이 큼
Dynamic Creation을 통해 균등분할해줘야함

고려할 점
고난위도 thread 컨트롤이 필요 (모듈제작시)
(CUDA의 자동 thread 생성기능이 오히려 속도저하를 불러올 수 있음)
Kernel수준의 SMT코딩 필요, Global 함수에서 통합관리

We need static variables but cuda do not support !!!

Case II 분석

진행상황 – 현재 작업 진행중 (디버그중) : 현재 전체 코드 491 line
모듈화 작업을 병행하다보니 코드 길이가 길어짐

SPMD 1 방법을 이용했음 (모듈화 용이)
GPU에서 single code를 실행시켰을 경우 CPU대비 1.8배 느림
이론상 하나의 보드에서 약 3-40배 정도 성능향상 기대됨
현재 코드에서 약 12배 가속됨

현재 코드 전체를 변경할 예정
SPMD 2 방법을 이용하고 Memory공유시 SPMD1에 비해 4배 속도향상 기대됨

이외에 몇가지 속도 향상 기법이 존재 대부분 메모리, thread 관리 기법임.

왜 128개의 SP를 쓰는데 128배가 안나오지?

이론상
128개의 SP가 128개의 SFU를 가지고 있지 않고 32개의 SFU만 가짐.

GPU : 500Gflops, 최고성능 CPU : 40 Gflops : 약 40배 차이
일반성능 CPU 5 Gflops :약 100배 차이

일반 CPU 1core와 GPU 전체와 비교하면 100배 정도 성능 차이가 남

GPU의 clock speed가 CPU의 clock speed보다 떨어지기 때문에
GPU의 1개의 SP와 CPU 1개의 Core를 비교하면 CPU core 1개의 성능이 더좋음

따라서 GPU의 1개 core대비 SP의 성능 0.4~0.8 * 128 = 약 40~80배 정도
성능 향상 효과가 있음.

최초 출시되었을 때 CPU대비 100배 속도 상향이 가능했지만,
현재는 40배 정도 향상시키면 최적코드로 생각됨

multiGPU 이용시 N배의 scailability 보장함 (MC 방법론의 경우)

결론

•Tesla C870을 이용하여 2sec 안에 ELS price 계산 가능?

•즉, 100배 이상 속도 향상 가능?

C870 하나의 보드 이용시

•병렬화시 overlapping, cyclic 문제가 발생할 수 있는

•LCG를 이용시

•RNG 생성시 112배 빨라짐

•전체 프로세스에서는 **80배** 향상됨 (업무적으로 쓰기에는 부적합할 것으로 생각됨.)

• MT19937 이용시,

•현재 **12배** 정도 가속됨. (최적화 안되었음, 계속적인 연구 중)

•조기종료를 고려할 경우에도 CPU조기종료 대비 약 12배 정도(약간씩 오차있음) 빠름

결론

CUDA를 이용한 MC 병렬화 (MT19937 이용시)

1. 현재 12배, 좀더 최적화하면 ELS pricing 속도 약 20배 향상 예상
   - 1개의 보드 장착 200sec ➔ 17.00 sec (현재)
   - 1개의 보드 장착 200sec ➔ 10.00 sec (최적화중)

2. multiGPU 시스템 연구예정
   - 4개의 보드 장착된 S870 1대 이용 200sec ➔ 2.50 sec (예상) 80배
   - 4개의 보드 장착된 S870 2대 이용 200sec ➔ 1.25 sec (예상) 160배

3. shared memory size의 제한으로 고속화 불가
   - block당 2.7 KB 가능하다면 약 40배 속도 향상 200sec ➔ 4.00 sec (예상)

LCG48 이용시 1개의 보드에서 80배 성능향상 보장

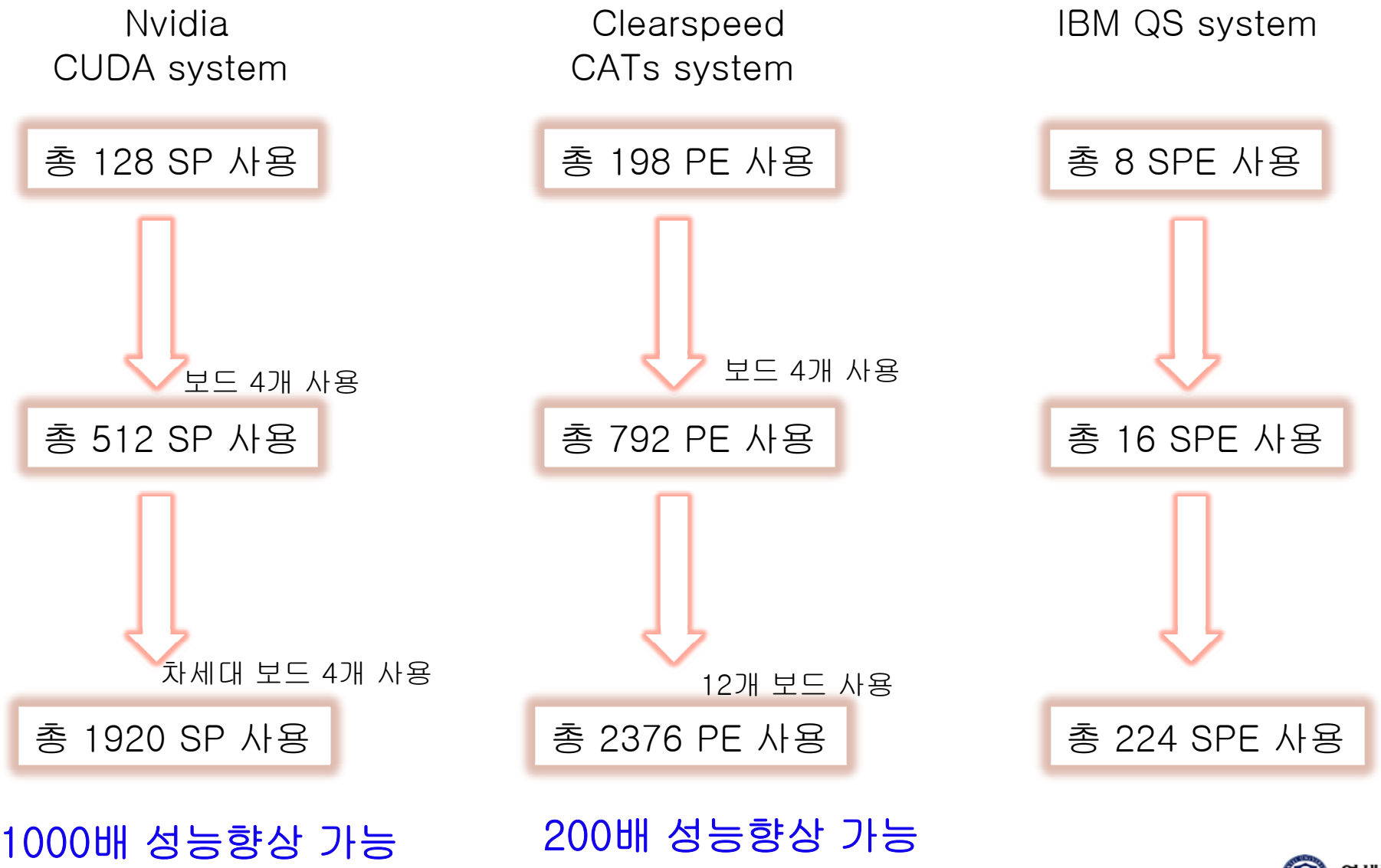
연구 계획

향후 전망 (CUDA를 이용한 금융)

현재 : 100개 상품을 9000sec = 2.5시간에 계산함

곧 실현될 미래 : 100개 상품을 2sec 만에 계산함

Further Research II CATs system & IBM Cell



1000배 성능향상 가능

200배 성능향상 가능

The End

경청해 주셔서 감사합니다.

연세대 수학과 박사과정 유현곤
E-mail : yhgon@yonsei.ac.kr